1993

# An applicative framework for VLSI programming.

Dimitris. Phoukas
*University of Windsor*

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

## NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.



## AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.



Canada

# An Applicative Framework for VLSI Programming

by

**Dimitris Phoukas**

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the Department of Electrical Engineering
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy of Electrical Engineering at the
University of Windsor

Windsor, Ontario, Canada
1992

ISBN 0-315-83076-X

Name Dimitris Phoukas

*Dissertation Abstracts International* is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science
SUBJECT TERM

0984
SUBJECT CODE

U·M·I

## Subject Categories

# THE HUMANITIES AND SOCIAL SCIENCES

### COMMUNICATIONS AND THE ARTS
- Architecture 0729
- Art History 0377
- Cinema 0900
- Dance 0378
- Fine Arts 0357
- Information Science 0723
- Journalism 0391
- Library Science 0399
- Mass Communications 0708
- Music 0413
- Speech Communication 0459
- Theater 0465

### EDUCATION
- General 0515
- Administration 0514
- Adult and Continuing 0516
- Agricultural 0517
- Art 0273
- Bilingual and Multicultural 0282
- Business 0688
- Community College 0275
- Curriculum and Instruction 0727
- Early Childhood 0518
- Elementary 0524
- Finance 0277
- Guidance and Counseling 0519
- Health 0680
- Higher 0745
- History of 0520
- Home Economics 0278
- Industrial 0521
- Language and Literature 0279
- Mathematics 0280
- Music 0522
- Philosophy of 0998
- Physical 0523
- Psychology 0525
- Reading 0535
- Religious 0527
- Sciences 0714
- Secondary 0533
- Social Sciences 0534
- Sociology of 0340
- Special 0529
- Teacher Training 0530
- Technology 0710
- Tests and Measurements 0288
- Vocational 0747

### LANGUAGE, LITERATURE AND LINGUISTICS
- Language
  - General 0679
  - Ancient 0289
  - Linguistics 0290
  - Modern 0291
- Literature
  - General 0401
  - Classical 0294
  - Comparative 0295
  - Medieval 0297
  - Modern 0298
  - African 0316
  - American 0591
  - Asian 0305
  - Canadian (English) 0352
  - Canadian (French) 0355
  - English 0593
  - Germanic 0311
  - Latin American 0312
  - Middle Eastern 0315
  - Romance 0313
  - Slavic and East European 0314

### PHILOSOPHY, RELIGION AND THEOLOGY
- Philosophy 0422
- Religion
  - General 0318
  - Biblical Studies 0321
  - Clergy 0319
  - History of 0320
  - Philosophy of 0322
- Theology 0469

### SOCIAL SCIENCES
- American Studies 0323
- Anthropology
  - Archaeology 0324
  - Cultural 0326
  - Physical 0327
- Business Administration
  - General 0310
  - Accounting 0272
  - Banking 0770
  - Management 0454
  - Marketing 0338
- Canadian Studies 0385
- Economics
  - General 0501
  - Agricultural 0503
  - Commerce-Business 0505
  - Finance 0508
  - History 0509
  - Labor 0510
  - Theory 0511
- Folklore 0358
- Geography 0366
- Gerontology 0351
- History
  - General 0578
  - Ancient 0579
  - Medieval 0581
  - Modern 0582
  - Black 0328
  - African 0331
  - Asia, Australia and Oceania 0332
  - Canadian 0334
  - European 0335
  - Latin American 0336
  - Middle Eastern 0333
  - United States 0337
- History of Science 0585
- Law 0398
- Political Science
  - General 0615
  - International Law and Relations 0616
  - Public Administration 0617
- Recreation 0814
- Social Work 0452
- Sociology
  - General 0626
  - Criminology and Penology 0627
  - Demography 0938
  - Ethnic and Racial Studies 0631
  - Individual and Family Studies 0628
  - Industrial and Labor Relations 0629
  - Public and Social Welfare 0630
  - Social Structure and Development 0700
  - Theory and Methods 0344
- Transportation 0709
- Urban and Regional Planning 0999
- Women's Studies 0453

# THE SCIENCES AND ENGINEERING

### BIOLOGICAL SCIENCES
- Agriculture
  - General 0473
  - Agronomy 0285
  - Animal Culture and Nutrition 0475
  - Animal Pathology 0476
  - Food Science and Technology 0359
  - Forestry and Wildlife 0478
  - Plant Culture 0479
  - Plant Pathology 0480
  - Plant Physiology 0817
  - Range Management 0777
  - Wood Technology 0746
- Biology
  - General 0306
  - Anatomy 0287
  - Biostatistics 0308
  - Botany 0309
  - Cell 0379
  - Ecology 0329
  - Entomology 0353
  - Genetics 0369
  - Limnology 0793
  - Microbiology 0410
  - Molecular 0307
  - Neuroscience 0317
  - Oceanography 0416
  - Physiology 0433
  - Radiation 0821
  - Veterinary Science 0778
  - Zoology 0472
- Biophysics
  - General 0786
  - Medical 0760

### EARTH SCIENCES
- Biogeochemistry 0425
- Geochemistry 0996
- Geodesy 0370
- Geology 0372
- Geophysics 0373
- Hydrology 0388
- Mineralogy 0411
- Paleobotany 0345
- Paleoecology 0426
- Paleontology 0418
- Paleozoology 0985
- Palynology 0427
- Physical Geography 0368
- Physical Oceanography 0415

### HEALTH AND ENVIRONMENTAL SCIENCES
- Environmental Sciences 0768
- Health Sciences
  - General 0566
  - Audiology 0300
  - Chemotherapy 0992
  - Dentistry 0567
  - Education 0350
  - Hospital Management 0769
  - Human Development 0758
  - Immunology 0982
  - Medicine and Surgery 0564
  - Mental Health 0347
  - Nursing 0569
  - Nutrition 0570
  - Obstetrics and Gynecology 0380
  - Occupational Health and Therapy 0354
  - Ophthalmology 0381
  - Pathology 0571
  - Pharmacology 0419
  - Pharmacy 0572
  - Physical Therapy 0382
  - Public Health 0573
  - Radiology 0574
  - Recreation 0575
  - Speech Pathology 0460
  - Toxicology 0383
- Home Economics 0386

### PHYSICAL SCIENCES
#### Pure Sciences
- Chemistry
  - General 0485
  - Agricultural 0749
  - Analytical 0486
  - Biochemistry 0487
  - Inorganic 0488
  - Nuclear 0738
  - Organic 0490
  - Pharmaceutical 0491
  - Physical 0494
  - Polymer 0495
  - Radiation 0754
- Mathematics 0405
- Physics
  - General 0605
  - Acoustics 0986
  - Astronomy and Astrophysics 0606
  - Atmospheric Science 0608
  - Atomic 0748
  - Electronics and Electricity 0607
  - Elementary Particles and High Energy 0798
  - Fluid and Plasma 0759
  - Molecular 0609
  - Nuclear 0610
  - Optics 0752
  - Radiation 0756
  - Solid State 0611
- Statistics 0463

#### Applied Sciences
- Applied Mechanics 0346
- Computer Science 0984
- Engineering
  - General 0537
  - Aerospace 0538
  - Agricultural 0539
  - Automotive 0540
  - Biomedical 0541
  - Chemical 0542
  - Civil 0543
  - Electronics and Electrical 0544
  - Heat and Thermodynamics 0348
  - Hydraulic 0545
  - Industrial 0546
  - Marine 0547
  - Materials Science 0794
  - Mechanical 0548
  - Metallurgy 0743
  - Mining 0551
  - Nuclear 0552
  - Packaging 0549
  - Petroleum 0765
  - Sanitary and Municipal 0554
  - System Science 0790
- Geotechnology 0428
- Operations Research 0796
- Plastics Technology 0795
- Textile Technology 0994

### PSYCHOLOGY
- General 0621
- Behavioral 0384
- Clinical 0622
- Developmental 0620
- Experimental 0623
- Industrial 0624
- Personality 0625
- Physiological 0989
- Psychobiology 0349
- Psychometrics 0632
- Social 0451

# ABSTRACT

In the last few years there has been a growing interest in language-based hardware design and analysis. Unfortunately, the choice of a linguistic framework is still a major issue. In this thesis we argue that an applicative, wide-spectrum linguistic framework in which specifications as well as implementations can be expressed is the best candidate for this task. A higher order, strongly typed **Applicative System Description Language** (ASDL) in which only well founded recursion can be described has been designed and implemented in order to demonstrate these principles in language design.

The main contribution of this work is that, within a single, architecture-independent framework, a variety of design concerns (views) can be expressed and formally manipulated. Therefore, this thesis offers additional evidence for the fact that VLSI design may be regarded as a kind of applicative programming. This is important because we can expect that the productivity gains exhibited by applicative programming techniques can be transferred to the field of VLSI design. It is also expected that, since the formal methods used in applicative programming help to construct better programs, they will help us in building "better" circuits. But the greatest hope is that that it will be possible to synthesize circuits from specifications thereby making possible the application of the modern engineering philosophy "design correct first time".

In this thesis, a number of design idioms based on the Bird and Meertens Formalism and their generalization to temporal attribute grammars are proposed. At the switch-level, a formal model of MOS transistor behavior is used to justify a synthesis procedure of CMOS networks from incomplete boolean specifications under certain constraints about the operating environment. The approach is illustrated with a number of example derivations.

*To my parents, Alexander and Stavroula,*
*and my sister Christina, for believing in me.*

# Acknowledgments

I am in debt to both of my co-chairs for their advice and support during my doctoral studies. From predicate logic and knowledge bases to Montague semantics and attribute grammars, Richard Frost has been an excellent teacher and a great motivator for me. I hope that his perspective is reflected here in a fair light. Graham Jullien's infectious enthusiasm and ability to nurture emerging themes in VLSI design have been a constant source of strength and inspiration. I would also like to thank the members of the committee, Professors Ahmadi, Miller and Bandyopadhyay, for their participation and valuable comments.

My first roommates in Windsor, Panneer Gopal and Amir Mazinani, kept me calm at times when I found doing a Ph.D. to be a great strain. Bruce Erickson shared with me his understanding of CMOS technology during our late night visits to the Gradhouse. Peter Tsin helped me overcome frustration and anger at one of the most critical phases of my study. Steve Karamatos was the most friendly and responsive system administrator I could ever have wished for!

John Stefanides and Walid Saba along with WCSX kept me company during the early morning hours at the Grad Lab. My most recent office-mates, Arindam Das and Farook Wadia, have enlightened me with their knowledge of Indian culture. Special thanks must go to the members of the "Systolic Compiler" group, Arunita Sarkar, Wai Fong, Tibor Toronyi and Hien Bui.

It is somehow saddening to write that Astero Zolota, who spent three years of life in this endeavor, cannot share the joy of its completion. Marina Petroula has been my muse for a large part of my adult life. Finally, I am grateful to you, Anna and Sevi, for keeping me alive during the last and most difficult year of my studies.

# TABLE OF CONTENTS

# List of Figures

# List of Tables

# Chapter 1 INTRODUCTION

Engineering may be described as the application of scientific laws to the design and implementation of systems. Laws allow an engineer to reason about models of the system, thereby making possible the transition from a specification to an implementation that satisfies its requirements. The more accurate the models of the system, and the better the methods for reasoning about them, the better the software tools we can build to support the design process, and the less the need to resort to trial and error. In this thesis, we introduce a new notation for describing (models of) VLSI systems and a new framework for reasoning about them.

## 1.1 The Thesis

**The thesis statement:**

I. VLSI design may be regarded as a form of applicative programming.

II. The formal methods that have been developed for the applicative programming paradigm can be adapted and used to advantage in VLSI design.

**Why the thesis is important:**

The advances in the fabrication of integrated circuits have led to a situation in which it is now possible to produce complex systems in high volume featuring small size, high speed, cheap material and low power consumption. Existing approaches to VLSI design are inadequate in a number of respects. These inadequacies are discussed in detail in chapter 2. It is our belief that the same techniques that have been used in order to overcome the "software crisis" can be applied to master similar problems in hardware design. Establishment of the thesis should encourage a

technology transfer from the field of formal software development to the domain of circuit design, resulting in the formalization of the VLSI design proces. But the greatest hope is that it will be possible to synthesize circuits from specifications thereby making possible the application of the modern engineering philosophy "design correct first time", under the assumption that specifications are "accurate".

**The approach that has been used to substantiate the thesis:**

I. In order to demonstrate that VLSI design can be regarded as a form of applicative programming, the following work was undertaken :

   a. We designed a general purpose, strongly typed, higher order Applicative System Description Language in which only finite nets of systems can be described.

   b. We extended the applicative programming paradigm by developing various interpretations pertaining to different aspects of VLSI systems.

   c. We used the language in the design of a number of circuits.

II. In order to show that formal methods, developed for the applicative programming paradigm, can be adapted and used to advantage in VLSI design, the following work was undertaken :

   a. We developed new circuit design strategies, based on formal methods from applicative programming and tested them in various example VLSI designs.

   b. We developed software tools to support the design process and tested them in a number of example derivations.

## 1.2 Specific tasks undertaken

**Survey of related work**

Owing to the multi-disciplinary nature of the thesis it was necessary to undertake a substantial survey of literature from various domains in both Computer Science and VLSI design. In

particular, we surveyed work on functional programming (implementation and transformational derivation of programs), type systems and their use in the development of programs, attribute grammars, formal specification languages, formal hardware verification, functional geometry, hardware description languages, design methodologies for parallel architectures with emphasis on systolic arrays and switch-level modeling of MOS transistors.

**Development of a novel hardware description language**

We defined syntax and static elaboration rules for a new general purpose hardware description language (ASDL) with a novel construct for defining recursively defined circuits with guaranteed termination. We have defined the structural semantics of ASDL in terms of a translation to an intermediate wirelist language with object semantics (also designed by the author). Three behavioral semantics, static, stream and timing, have been defined, the third one (timing) employing a novel "lifting" from the static behavioral semantics. Finally, a modern type system has been designed allowing the expression of powerful data abstractions.

**Partial definition of a VLSI design style**

We defined, in ASDL, two sets of combinators related to particular hardware configurations, one for linear arrays and the other for tree-like architectures, and showed how thes can be generalized to any architecture. We developed and proved general theorems relating these combinators and discussed how these can be extended to other architectures. For dealing with bi-directional data flow, we recognized the value of regarding circuits specifications as attribute grammars and developed a methodology to translate ASDL descriptions into (from) a form of attribute grammars. We have defined and proved two generic theorems relating attribute grammars and a "retiming procedure" for circuits expressed as attribute grammars. Finally, we have tested the approach with the formal derivation of two examples, the parallel prefix operator and the carry lookahead adder.

**Definition of CMOS transformation laws**

We defined ASDL models of CMOS transistors incorporating a quality metric and proved laws relating two levels of hardware description, boolean gates and transistors. Based on these models and laws, algorithms for synthesizing static CMOS circuits from complete and incomplete specifications are given. The algorithms were tested with a number of examples including multiplexors and barrel shifters.

**Implementation of software to support use of ASDL in VLSI design**

Software tools were written to evaluate the use of our language in VLSI design. Most notable among them is the ASDL analyzer and expander, a simulator, a translator to VHDL and a boolean to CMOS circuit synthesizer.

## 1.3 Summary of important results

In this section we summarize the major contributions of the thesis which are discussed more thoroughly in section 8.2:

1. Our work generalizes both the first order applicative and the combinatory styles of hardware description in a unified framework.
2. Our notation allows the definition of strongly typed, (higher order) inductive combinators whose termination is guaranteed by the syntax.
3. We have successfully interpreted certain applicative programming styles (theory of lists, attribute grammars) as circuit design idioms. The last one, attribute grammars, allows reasoning about bidirectional data flow.
4. We have successfully modelled boolean to CMOS abstraction constraints in the applicative framework.

These results give us confidence that the formal methods that have been developed in the applicative programming domain can be transferred to the field of VLSI design and, therefore, deserve further attention from researchers in the field.

## 1.4 Organization of the thesis

In chapter 2, after reviewing the limitations of existing imperative hardware description languages, and the drawbacks of traditional methods for defining semantics, we discuss an alternative basis for defining the semantics of hardware description languages, namely "system semantics". This is the method used in chapter 3, to define the semantics of our system description language, ASDL. The unique features of the language, inductive definitions and isomorphic types, are also discussed in chapter 3.

In the following three chapters, the elements of a design calculus using ASDL are introduced. In chapter 4, we show how generic system combinators, "homomorphisms" and "paramorphisms", can be defined on any data type. We also prove a general law that allows systems to be "fused" inside a homomorphism or a paramorphism. The issue of bidirectional data flow is the subject of chapter 5 where an intuitive interpretation of ASDL descriptions as attribute grammars is used in order to derive new fusion laws and a general retiming theorem for the pipelining of inductively defined systems. The derivation of the parallel prefix operator and the carry-lookahead adder are also shown as example applications of this method. Switch-level models and transformation laws for CMOS transistors are given in chapter 6, along with algorithms for the formal synthesis of CMOS circuits from (incomplete) boolean specifications.

In chapter 7, the software tools implemented as part of the thesis work are briefly demonstrated. Finally, in chapter 8 related work is discussed and some directions for future research are proposed.

# Chapter 2 BACKGROUND

The great majority of existing Hardware Description Languages (HDLs) are based upon traditional, i.e. imperative semantics. This can be attributed to historical factors — the first formal languages were developed within the imperative framework which still is the dominant one in the realm of programming languages. It can be also attributed to the fact that traditional HDLs have been designed with the view that the main design validation method would be simulation and not symbolic manipulation. One can understand this approach taken by the E-CAD community in light of the relative success of design analysis using analog and gate simulation models in the late 70's.

Unfortunately, simulation was not able to cope with the increasing size and complexity of designs in the 80's. Simulation is non-hierarchical: the assumption that all inputs are stable and driven does not hold when we decompose a large system to its sub-parts [133]. It fails to "scale-up" with an increase in the number of inputs-outputs (no induction principle is applicable). Also, paraphrasing Dijkstra's famous aphorism for software testing, simulation indicates the presence of errors but not their absence or their location. Finally, exhaustive simulation of sequential circuits is frequently not even possible because not only the test vector itself but the order of applying stimulus matters (due to charge storage effects).

The four major concerns in today's hardware verification and synthesis formalisms are [93]:

1. **Structural abstraction:** internal composition of a system.
2. **Behavioral abstraction:** input-output mapping and environment constraints.

3. **Data abstraction**: (partial) mapping from implementation—oriented data types to more abstract data types.
4. **Timing abstraction**: relating two different "grains" of time. For example, at the register transfer level a time unit corresponds to the intervals between synchronizing clock events where at the timing level, a time unit corresponds to some interval of real time.

In the following three sections in this chapter we will indicate *(i)* why imperative semantics are inappropriate for dealing with each one of these concerns, *(ii)* why applicative formalisms are intrinsically better equipped to cope with these tasks and *(iii)* how "system semantics" is more appropriate for the description of systems that are not necessarily computational than the more traditional "denotational semantics" approach.

## 2.1 Limitations of existing hardware description languages based on imperative semantics

We begin by introducing some definitions taken from [21]:

An *effective procedure* is a finite, unambiguous description of a finite set of operations[1]. An *algorithm* is an effective procedure that terminates after a finite number of steps. A *program* is an effective procedure expressed in some appropriate formal (programming) language.

A *computational system* is a program together with an interpreter for the language in which the program has been expressed. A *computational process* is the dynamic behavior of a computational system. A *computation* is a computational process that terminates. The Theory of Computation refers to *discrete computations* only, i.e. to the evaluation of functions from one countable set to another. One way of describing a discrete computation is in terms of histories of discrete time-value pairs.

---

[1] Actually, this "definition" appeals to intuition rather than formality. A more formal definition could be given in terms of the Church-Turing thesis, but this is outside the scope of this investigation.

The following remarks do not refer to a particular traditional HDL, but rather, to their common properties with respect to the major hardware abstractions of interest:

1. Behavior is expressed procedurally, i.e. in terms of an imperative algorithm. As far as we are concerned, behavior expressed in this way does not constitute an abstract specification, it is just a concrete realization, albeit in a different direction, that of *programmed implementations*. It must be also noted that a large class of systems, the analog ones, cannot be described in terms of effective computational procedures. For example, analog feedback cannot be described as recursion in data [16].

2. Even in the case that it makes sense to model behavior in terms of discrete computations (synchronous systems is one example), the modelling process is extremely *operational* in nature. As an example, one cannot comprehend a VHDL behavioral block unless one has a thorough understanding of the event-driven algorithm currently used in the IEEE 1076 standard definition for the scheduling of the implied simulator actions.

3. Structure, which is inherently static in nature, cannot be described effectively in a procedure-oriented way. The use of control variables (which do not correspond to any design objects), assignment and repetition to describe multiple instantiations (like the use of "generate" statements in VHDL loops) support this view.

4. Data abstraction is, in a sense, irrelevant for imperative constructs like commands. Only the sub-language of expressions within a typical HDL can have a type.

5. Reasoning in the imperative paradigm cannot be performed within the language itself but requires a separate deductive system instead, i.e. "axiomatic semantics". For example, relating two levels of time in VHDL (abstraction from timing to register transfer level) can be expressed only as an assertion on attributes of a signal during simulation. Therefore, we conclude that the manipulative qualities of imperative HDLs are very weak.

It is for these reasons that we believe that traditional HDLs are not system description languages but, rather, source languages for "universal" simulators.

## 2.2 Advantages of applicative semantics with respect to hardware description

The problems encountered with imperative hardware description languages, as discussed above, have motivated our quest for more declarative formalisms. Among them, we have found that applicative formalisms [71] offer the most promising match between linguistic constructs and the system properties of interest. In particular:

1. A function, i.e. a black box, is probably the most abstract form of deterministic behavior. No extra synchronization primitives, like semaphors or monitors, are needed in order to express parallelism. No explicit storage plans, reflecting a particular von-Neuman view of memory, are required. The question of non-determinism has been recently addressed in the realm of functional programming languages [102] and it should be possible to transfer these results to the domain of system description languages as well.

2. Structure description is an instance of the module interconnection problem for which many applicative solutions exist in the literature; our language, presented in the next chapter, is one of them. The artificial dichotomy between control and data flow, existing in hardware description languages with imperative heritage, disappears in applicative formalisms.

3. The study of data types, which play a dominant role in hardware design as we will show in subsequent chapters, is essentially performed within the applicative framework. This makes the prospect of technology transfer to applicative system description languages very high.

4. Finally, both static and dynamic system properties can be described within the applicative framework, as will be demonstrated in the next chapter.

In addition, we note that reasoning in the applicative style is performed in the same framework as the language itself. It is transformational in nature (as opposed to deductive)[2], in line with current engineering practice [18]. At this point, we briefly introduce the proof techniques used in the rest of this thesis, "equational reasoning" and "structural induction".

The **equational** method of reasoning about descriptions in applicative form has its origins in the reduction mechanism by which the meaning of an expression is calculated: it is controlled, symbolic evaluation of the expression. In the context of transformational derivation, *reduction*, i.e. substituting the right-hand-side of an equation in place of the left-hand-side, is sometimes called *unfolding*. The reverse operation, using an equation from the right-hand-side to the left-hand-side, is called *folding* and it is not guaranteed to preserve termination, although in this work care has been taken so that this issue does not arise by using non overlapping definitions. The introduction of local definitions ("where" clauses) is called *abstraction*. In the following chapters, in equational derivations we will make use of a notation first introduced by Dijkstra where the justification for each step is enclosed in curly brackets and the particular equation used is denoted by the combination of definition name and equation number. Dots as in { . . . } will be used to indicate steps that are omitted as trivial.

Next, the most powerful technique for reasoning about applicative expressions is presented. It is based on the principle of induction over natural numbers. In analogy with natural numbers, any structure with a *well-founded ordering* can be used. A well-founded ordering is a relation (<) on a set $A$, such that $A$ contains no infinitely long decreasing chain of elements $...\alpha_3<\alpha_2<\alpha_1<\alpha_0$. The ordering is potentially partial. Using it, the **principle of total structural induction** is formulated:

---

2 Another way to say that is that reasoning follows a linear instead of a tree-like development.

**To prove that:** $\Phi(\chi)$ for all $\chi$ in $A$, where $<$ is a well-founded ordering on A, show the following:

**Base case:** For all *minimal* elements $\chi$ of $A$, i.e. for all $\chi$ in $A$ such that there is no other $\chi'<\chi$, $\Phi(\chi)$ is true.

**Inductive step:** Assuming that for all $\chi'<\chi$, $\Phi(\chi')$ (*inductive hypothesis*), prove $\Phi(\chi)$.

All data types definable in ASDL have the well-founded order property, therefore the principle of total structural induction is applicable to them. We will not make use of partial structural induction in this thesis.

## 2.3 System Semantics

Most physical systems in existence today are inherently non-computational, and hence the formalisms developed in order to give meaning to programs, e.g. denotational semantics, do not provide an adequate foundation for system description languages. Physical systems obey the laws of physics. The notion of a system executing a "hardware algorithm" as used by several authors [118] is in itself an oxymoron! Therefore, definition of a system description language in terms of denotational semantics is inappropriate.

An alternative to denotational semantics, called **system semantics** has been introduced by Boute [17] for the description of non-computational systems. System semantics is the description of system properties by means of a total meaning function $m$ mapping elements from $S$ (the set of sentences of our system description language) into elements of a domain of interpretation $D$ (the set of values that the property may assume). The meaning function together with the specific domain of interpretation constitute a **model** $M=(m,D)$ for the particular property of interest. The models themselves are considered part of the formalism. This particular viewpoint permits the

extension of the models to any calculi, e.g. cost, testability etc, provided that the appropriate domain of interpretation as well as the appropriate meaning functions are defined. Since meaning is described by means of a total function, a model completely specifies the semantics of the language and therefore is considered its primary definition.

**Semantic equivalence** with respect to a model $M$ is a relation $==_M$ defined by $r ==_M s <=> m\ r = m\ s$. Among semantic equivalences, the most interesting is behavioral equivalence. When the model is obvious from the context, the subscript will be omitted as in chapters 4 and 5 where $M$ is the implied stream behavioral model. We say that a model $M$ provides at least as much detail as a model $M'$ if equality in $M$ implies equality in $M'$ as well. The model that provides the more detail than any other is called the **initial system model**. The level of detail provided by a specific system description language is (by definition) that of its initial system model[3].

**Syntactic equivalence** is defined by taking the reflexive, symmetric and transitive closure of a set of conversion rules relating elements of syntactic categories. A set $R$ of conversion rules is called:

- **Consistent** (or sound) with respect to a model $M$ if $==_R$ implies $==_M$. Consistency is an essential property of any set of conversion rules.
- **Complete** with respect to a model $M$ if $==_M$ implies $==_R$.

In contrast with the usual axiomatic approach, the conversion rules do not complement the semantic definition, but are merely consequences of it.

## 2.4 Concluding comments

In this chapter, the limitations of hardware description languages based on imperative semantics have been discussed and the use of applicative notation has been contemplated. It should be noted that, although directionality is not a property of the system to be described but, rather, of

---

[3] One way to think about system semantics is as a generalization of "abstract interpretation" in the direction of more concrete domains.

the formalism used for its behavioral analysis, it has become such a pervasive characteristic of modern digital systems that it can be usefully regarded as part of the interface even when other system aspects are considered. In the area of digital signal processing in particular it is assumed that the underlying systems are uni-directional and that the output signal is not affected by loading. For this class of systems the (spartan) syntax of $\lambda$—calculus has long been recognized as being a fitting basis for designing system description languages. Unfortunately, this is not the full story since the extremely uniform syntax of $\lambda$—calculus does not discriminate between functions and data with the result that no reasonable notation for resource sharing, which is of paramount importance in hardware description, is available. Also, the way in which iteration is handled in $\lambda$—calculus (unrestricted recursion) is far from satisfactory. We will demonstrate these problems and our solutions to them with the design and implementation of our Applicative System Description Language which is described in the following chapter.

# Chapter 3 AN APPLICATIVE SYSTEM DESCRIPTION LANGUAGE

In this chapter we present the design rationale and implementation decisions for an Applicative System Description Language, which we shall refer to from now on as ASDL. The initial system model of ASDL has been chosen to be the structural one (enriched with "generic" parameters and user-defined data types) because, in our opinion, this is the most appropriate level for the design of today's VLSI systems and because the mechanisms of structural abstraction remain essentially the same across different levels of detail. More detailed models, e.g. floorplans, or layout, could be obtained from the structural model in two ways:

1. Automatic (or interactive) place and route, assuming that this technology will be able to keep up with the increasing densities and intricate complexities of modern VLSI and ULSI systems.
2. Adding relative placement annotations in the style of LTS [98]. It should be noted here that geometry is ideally suited for layout description languages with declarative semantics [34]. This is the case with the great majority of the existing layout description languages.

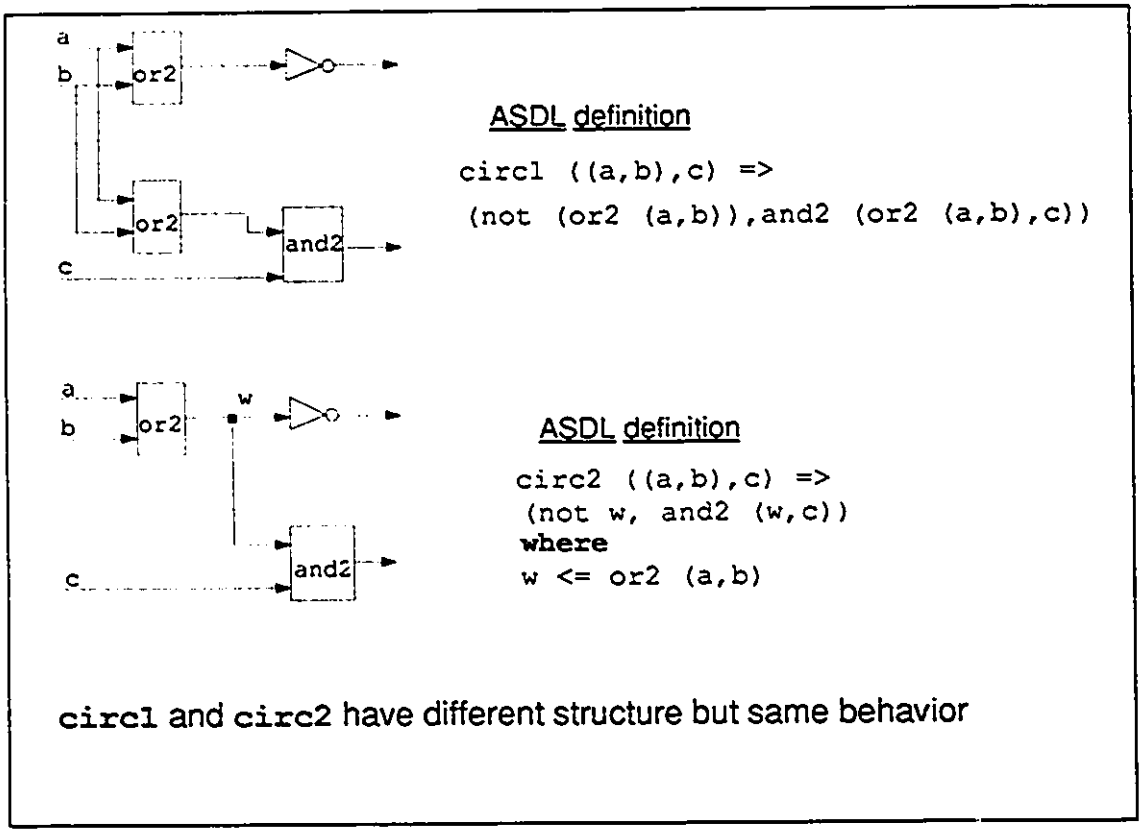
However, in this thesis we do not consider these more detailed models[1].

## 3.1 Introduction

The basic syntax of $\lambda$—calculus is a good starting basis for describing unidirectional systems, if we interpret function application, denoted by simple juxtaposition, as serial connection and

[1] An alternative approach would be to design a system description language whose initial semantics is abstract floorplans in the tradition of *vFP* [111]. Although this might seem a more rational choice for systems that, after all, have a planar implementation, nevertheless it has the definitive disadvantage that it forces the designer to confront geometry issues very early in the design cycle.

Figure 3.1.1. Replication vs. fanout



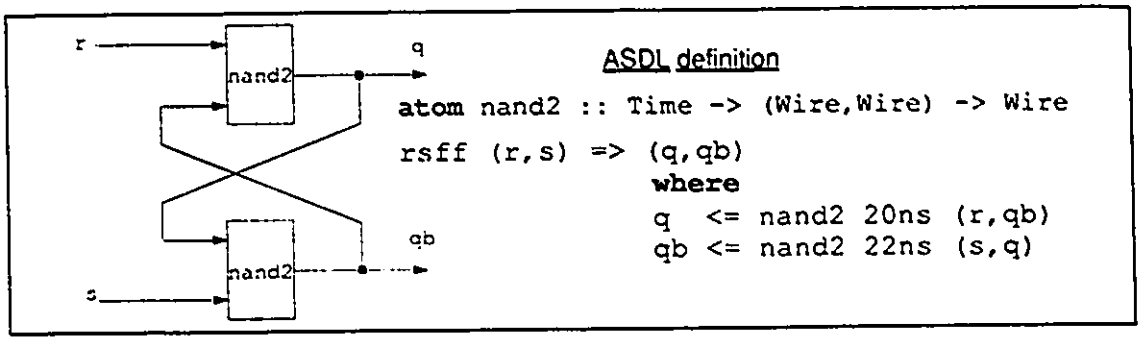

tupling as the formation of buses. Unfortunately, these simple mechanisms are inadequate in describing two fundamental circuit attributes: output sharing (fanout) and feedback. The solution is to introduce some syntactic sugar, already familiar to users of block-structured languages: local (data) bindings, and interpret them as textual representations of graphs. The idea was derived from the work on *graph term rewriting* which underlies the implementation of the lazy functional language Concurrent Clean [127]. The schematic[2] in figure 3.1.1 illustrates the difference between fanout and replication: repetition of variables in the same context denotes fanout (sharing of outputs) whereas repetition of other types of expressions denote replication of circuit structure. Arrows (=> and <=) indicate the direction of signal flow.

---

2 One way to think about ASDL definitions is as linear representations of schematic diagrams.

Local data bindings are allowed to be recursive, in which case they are interpreted as denoting feedback. The following example (figure 3.1.2) of a RS flip-flop illustrates this point. Notice that nominal real-time propagation delays have been assumed for the nand gates. These are examples of generic parameters of type Time.

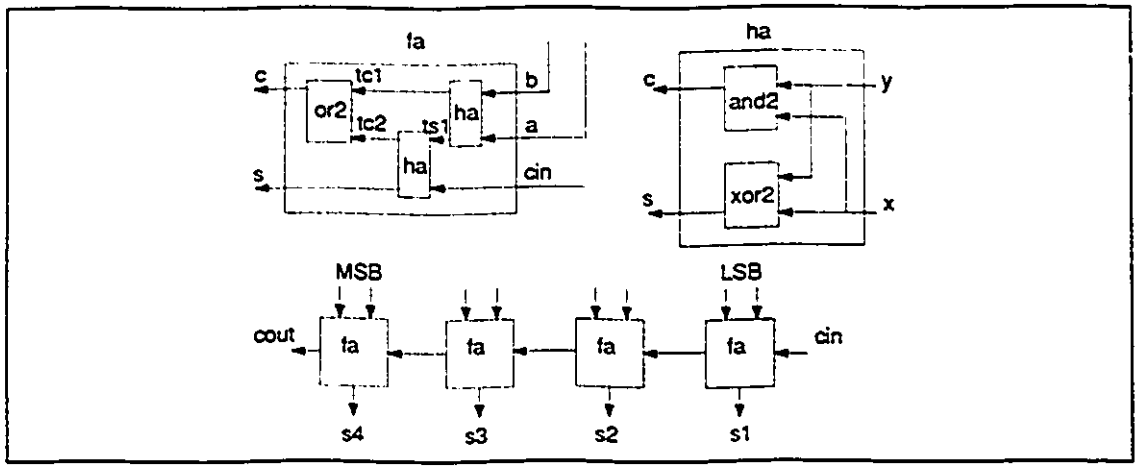
Figure 3.1.2. An example of recursion in data (feedback)




In many modern circuit design styles, we need to be able to describe iterative structure. A crucial characteristic of structure described in this way is that the resulting structures must be *static*, i.e. should not depend on any specific input values. Inductive definitions use a combination of pattern matching and a special syntactic form, inductive variables, that make it possible to describe circuit structure that depends on the "shape" of the input. *No direct or indirect recursion of system definitions is allowed.* As an example, one possible definition of a 4 bit binary adder as an instance in a family of ripple-carry adders which depends on the size of the input list, is given in figure 3.1.3.

```
basetype = Wire
/* This is a comment! Full-adder description follows */
fa :: (Wire,(Wire,Wire)) -> (Wire,Wire)
fa (cin,x) => (s,or2 (tc1,tc2))
              where
              (ts1,tc1) <= ha x
              (s,tc2)   <= ha (ts1,cin)
ha :: (Wire,Wire) -> (Wire,Wire) // Half-adder
ha a => (xor2 a,and2 a)
/* A generic ripple-carry adder */
adder_n :: Wire -> [(Wire,Wire)] -> (Wire,[Wire])
adder_n c []      => (c,[])
```

Figure 3.1.3. A ripple carry binary adder




```
adder_n c (a:<as) => (cout,b:<bs)
                     where
                     (b,cout) <= fa (cx,a)
                     (cx,bs)  <= rec as
/* A 4-bit adder */
adder4 :: (((Wire,Wire),(Wire,Wire),(Wire,Wire),(Wire,Wire)),Wire)
       -> (Wire,[Wire])
adder4 ((x1,x2,x3,x4),cin) => adder_n cin [x1,x2,x3,x4]
```

`Wire` is the type of the basic connection in electrical systems. The notation `(a:<as)` denotes the pattern that matches a finite list with prefix `a` and tail `as`. The expression **`rec`** `as` in the definition of the generic adder will be replaced with the recursive call `adder_n c as`. Since at each recursive step we only use a part of the original argument, and ASDL data types are finitely deep, this restricted form of recursion is well-founded and termination is guaranteed by the scope rules of the inductive variables (this argument cannot be made more formal unless we know more about the semantics of ASDL types). This is of indispensable value in transformational design, as we will find out in chapters 4 and 5. One can compare the conciseness and elegance of the above descriptions with a typical binary adder definition in VHDL which would involve the use of the "generate" command combined with conditional statements. Finally note that the (optional) type signatures for `fa` and `ha` reveal that these subcircuits have the appropriate type, although the input patterns are not fully unfolded. The static typechecking system of ASDL will

infer the correct types for them based on information about the types of the atomic components `or2`, `and2` and `xor2`. These are declared as follows:

```
atom or2  :: (Wire,Wire) -> Wire
atom and2 :: (Wire,Wire) -> Wire

atom xor2 :: (Wire,Wire) -> Wire
```

In some technologies, e.g. CMOS, it is meaningful to connect together two or more wires. This cannot be expressed in our pure applicative notation because it would imply that a single net would be driven by more than one sources. We solve this problem in ASDL with the use of **bus resolution functions**[3]. As an example, a "wired-or" connection of wires `xs` is expressed as:

```
join wor :: (Wire,Wire)->Wire

wired_or :: [Wire] -> Wire
wired_or xs => redr1 wor xs

redr1 op [a]     => a
redr1 op (a:<x) => op (a,rec x)
```

The systematic definition of generic combinators like `redr1` is discussed in the next chapter. In the remainder of this chapter, the syntax, the elaboration of inductive definitions and inductive variables, the type system and the semantics of ASDL are described in detail.

## 3.2 Syntax and scope rules

We will concentrate our attention now on a simple subset of ASDL where all system definitions are given at the top level. The context-free syntax of system definitions is given in figure 3.2.1 (the concrete syntax can be found in appendix A and data constructors are discussed in section 3.4.2).

---

[3] Another use of bus resolution functions (which is not explored in this thesis) is in describing bidirectional busses connecting systems with tristate outputs.

Figure 3.2.1. Context-free syntax of ASDL definitions

```
x,x1,...xn :: Variable (read as x,x1,...xn are
                        of type Variable)
p,p1,...pn :: Pattern
ip,ip1,...ipn :: Irrefutable_Pattern
C :: Constructor       (includes both sum
                        and product constructors)
P1,...Pn :: Product_Constructors
k :: Literal           (positive integers,time)
e,e1,...en :: Expression

def ::= x p1 ... pn => e
p ::= x
    | _                        (wildcards)
    | p1 :< p2
    | p1 :> p2
    | [p1, ... ,pn]            (n>=0)
    | k + x
    | (p)
    | (p1, ... ,pn)            (n>1)
    | C p1 ... pn              (C has arity n, n>=0)
    | k
ip ::= x
     | (ip1, ... ,ipn)         (n>1)
e ::= x
    | e1 e2                    (applications)
    | C
    | k
    | e1 :< e2
    | e1 :> e2
    | [e1, ... ,en]            (n>=0)
    | (e)                      (parenthesized expressions)
    | (e1, ... ,en)            (n>1)
    | rec x                    (inductive variables)
    | e where                  (local clauses with
        ip1 <= e1               irrefutable pattern
        ....                    bindings

        ipn <= en               where n>0)
```

Anonymous $\lambda$—abstractions and local system definitions not involving pattern-matching can be added to this spartan language at the expense of an additional pre-processing phase where simple, i.e. not fully-lazy, $\lambda$—lifting ([68], section 6.3) is performed. Note that no unit tuples () are allowed. The identities in figure 3.2.2, used as left-to-right rewrite rules, will be applied as

needed in the following discussion.

Figure 3.2.2. *TE translation scheme*

```
[e1,e2, ... en]  == e1:<(e2:<( ... (en:<[])))
e1 :< e2         == Cons e1 e2
e1 :> e2         == Snoc e1 e2
[]               == Nil
x + k            == k + x                 associativity of (+)
k + x            ==  Succ (Succ ...(Succ x)...)
                     (k-times Succ, k is a positive integer)
0                == Zero
(e1,..en)        == Pn e1 ... en          for all n>1
(e)              == e
```

In addition, the following scope rules apply:

1. A definition limits the scope of its formal parameters (the set of variable names bound by the definition's formal arguments). In addition, if a formal parameter x has the same type as the corresponding formal argument, then the variable **rec** x is also in scope.

2. The set of variable names bound by a local clause must not contain repetitions, i.e. every variable must have a unique binding occurrence at any given context. This corresponds to the design rule that no two or more outputs can be connected together (unless by explicit use of bus resolution functions).

3. Variable names within local clauses are local to them unless already bound by formal parameters, names of system definitions, names of atomic definitions or bus resolution function names. Note that these scoping rules are similar to the ones for let(rec) expressions as used in many functional languages.

Note that these scope rules leave open the possibility that a system output can be left "floating", i.e. not used as an input to any other system definition. Also, the following restrictions on patterns, the justification of which is discussed in section 3.3.2, must be enforced by any implementation of ASDL:

1. The set of patterns appearing as formal parameters in a definition must be *linear*, i.e. no variable may appear more than once in the set (unlike Miranda's™ pattern—matching conventions).

2. Patterns are *exhaustive*, i.e. cover all constructors of the type(s) involved.

Finally, the following restrictions on system definitions guarantees well-formedness:

1. System definitions can be nested but not (mutually) recursive.

2. If an inductive variable `rec` x appears at the right-hand-side of a system definition, then x must be a member of the set of formal parameters of this definition.

Note that the order of top-level system definitions as well as that of local bindings is irrelevant. In the following section, we will assume that all the above restrictions are satisfied and that system definitions are correctly typed.

## 3.3 Elaboration

The formal semantics of ASDL will be presented in terms of a conversion into a **kernel** language on which semantic functions for structure, behavior and other properties of interest can be applied. We call this process **elaboration**. It is *independent* of any particular model, i.e. the syntactic conversion rules are consistent (in the sense of section 2.3) with all models.

### 3.3.1 Floating of local definitions

The grammar for expressions given in section 3.2 allows nesting of local clauses to an arbitrary depth. This is helpful for system description purposes but complicates later translations. Therefore, during *scope analysis*, every identifier is assigned a unique name by systematic $\alpha$—conversion and, subsequently, all local definitions are moved leftward. We call this **floating**

of local definitions (it is similar to what is called $\lambda$—migration in [17]) because it results in a single local clause at the right-hand-side of a system definition. As an example, the expression:

```
(and2 (x1,r),r<:ys)
where
(r,ys) <= (and2 (x2,r),r:<ys)
          where
          (r,ys) <= (and2 (x3,r),r:<ys)
                    where
                    (r,ys) <= (xr,[])
```

is transformed by floating into:

```
(and2 (x1,r1),r1:<ys1)
where
(r1,ys1) <= (and2 (x2,r2),r2:<ys2)
(r2,ys2) <= (and2 (x3,r3),r3:<ys3)

(r3,ys3) <= (xr,[])
```

### 3.3.2 The semantics of pattern-matching and inductive variables

In order to facilitate the description of repetitive circuit structures, ASDL employs an innovative concept, **inductive system definitions**. These are defined by declaring a consecutive series of exhaustive but possibly overlapping equations which are tried in sequence from top to bottom. Within each equation, pattern-matching proceeds from left to right. The semantics of pattern-matching are described by first translating inductive definitions into a new syntactic form, "case expressions", which requires an extension of the previously defined grammar:

```
exp ::= ...
      | case e of {p1=>e1;...pn=>en}
```

Case expressions can be nested, in which case the usual scope rules apply. No guards are allowed in patterns.

Since local clauses employ irrefutable pattern bindings only, run-time **conformality checks** ([67], p. 115) are not required. Therefore, the only place where patterns can be refuted is at top level inductive system definitions. The following system of inductive definitions:

```
y p11 ... p1k => e1
...
y pn1 ... pnk => en
```

where n>=1, is equivalent to:

```
y x1 ... xk => case (x1, ... ,xk) of {
                    (p11,...,p1k) => e1;
                        ....
                    (pn1,...,pnk) => en}
```

and where x1,...xk are new variable names. This form will be taken as the starting point for the elaboration of pattern-matching.

The identities in figure 3.3.1 that define the semantics of all case expressions (modulo isomorphic types and data types with congruence rules) assume the *TE* rules, i.e. list and number patterns have been converted into patterns involving "sum constructors" (see next section) and tuples into product patterns.

Figure 3.3.1. Translation of pattern-matching *(TC translation scheme)*

```
(TC1) case e of {C p1...pn=>e1; _=>e2} ==
      case e2 of {
      y=> {case e of {
           C x1...xn=>case x1 of {
                      p1=>...case xn of {pn=>e1; _=>y}...
                      _ =>y}
           _=>y}
      where at least one of p1,..pn is not a variable and
      y,x1,...xn are new variable names
(TC2) case e of {x=>e1;_=>e2} == case e of {x=>e1}
(TC3) case e of {x=>e1} == e1[e/x]
      (substitute e for x in e1)
(TC4) case (C e1...en) of {C' x1...xk=>e1; _=>e2} == e2
      when constructor C is different from C' with
      arities n and m respectively
(TC5) case (C e1...en) of {C x1...xn=>e; _=>e'} ==
      case e1 of {
      x1'=>...case en of {xn'=>e[x1'/x1...xn'/xn]}...}
      where x1'...xn' are new variable names and
      C is a constructor of arity n
```

In words, pattern-matching an expression against a nested pattern (rule *TC1*) is equivalent to pattern-matching all its sub-patterns from left to right. Pattern-matching an expression against a

variable (rules *TC2*, *TC3*) always succeeds. Pattern-matching a composite expression against a composite pattern fails (rule *TC4*) if the constructors are different (or, in case of product patterns, if they have different arity); it succeeds (rule *TC5*) if their constructors are identical and their sub-patterns match. Note that we have imposed the constraint (enforced by the syntax rules) that patterns cannot be partial applications of constructors.

Finally, the following rule *(TR translation scheme)* gives the semantics of inductive variables: if `x` is a variable in the pattern `pi`, `1<=i<=n`, the case expression (E)

```
case e of {p1=>e1;...;pi=>ei(rec x);...pn=>en}
```

is rewritten into

```
case e of {p1=>e1;...pi=>ei(E[x/e]);...pn=>en}
```

Note that `ei` may contain variables bound by outer case expressions or (ultimately) formal parameters of the enclosing inductive system definition, which therefore become parameters of the case expression[4]. This is exactly the reason why the order of formal arguments in a system definition matters.

With the exception of isomorphic types, translation schemes TC and TR completely specify the semantics of pattern-matching in the presence of inductive variables for all case expressions (or, equivalently, for all inductive definitions). The elaboration of definitions with explicit coercions between isomorphic types is the subject of section 3.4.2.

When data types are defined in a mutual inductive mode, system definitions should be given in a mutual inductive mode as well. The types of inductive variables will be sufficient to discriminate the specific inductive function (see [25]). In this thesis though, we will not make use of mutually inductive data types.

---

[4] In chapter 5 we will show that parameterized inductive definitions correspond to a class of attribute grammars and that the parameters themselves play the role of inherited attributes.

## 3.4 The type system

ASDL is a strongly typed language. Every system definition is assigned a type signature that specifies the external interface of a system. They can be polymorphic in the sense that they can be parameterized by universally quantified type variables, with the usual restriction that these appear at the outside of a type. This is called **parametric polymorphism** [99]. Types in ASDL allow mixed-level description, i.e. a part of the system can be decomposed at its bit-level components whereas other parts can be described at more abstract levels. Data abstractions are expressed with the use of isomorphism declarations; their interaction with pattern-matching and inductive variables increases the expressible power of ASDL enormously. Typechecking is static and is performed using the standard Damas-Milner type inference algorithm [39] which infers the "principal type", i.e. the most general type, after inserting explicit coercions in place of type-isomorphisms and congruence rules. In the following sections the main features of the type system are described.

### 3.4.1 Base and physical types

Base types in ASDL are uninterpreted, i.e. they are simply placeholders in the type inference algorithm. Base type declarations introduce names for elementary connections and/or abstractions of them. In the domain of electrical systems, the elementary connection between subsystems is through wires, but other possibilities, like magnetic, optic, hydraulic, or pneumatic coupling exist in more general systems. It is conceivable that many of them can coexist in a mixed system. For example, in a design abstraction with mixed binary and integer data buses, the following declaration makes sense:

```
basetype = Int_wire, Bit_wire
```

Base types are declared at the top level only and are parameterless; successive base type declarations are equivalent to one which defines the union of the typenames introduced at the right-hand-side of the definitions.

At this point it should be noted that integers, which appear in the syntax of ASDL, are *not* assumed to be basetypes. They can be seen as providing syntactic sugar for defining projector functions on lists and other user-defined data types as well as convenience in the definition of physical types like `Time`[5]. Physical types are defined as variant types, tagged with their unit constructors and (optional) conversion rules between them. The following is the definition of a physical type, `Time`, where, for readability reasons, the constructors are given in postfix form:

```
data Time = Int fs | Int ps | Int ns | Int us
          | Int ms | Int sec | Int min
            where
            n min => (60 * n) sec
            n sec => (1000 * n) ms
            n ms  => (1000 * n) us
            n us  => (1000 * n) ns
            n ns  => (1000 * n) ps
            n ps  => (1000 * n) fs
```

The leftmost variant in the definition (`fs`) is assumed to be the basic time unit. In the interest of termination, the system of conversion rules must be noetherian and confluent. Unit conversions are instances of the so-called congruence rules which are introduced next.

### 3.4.2 Isomorphic types

We begin our description of the semantics of ASDL algebraic data types by an example. The built-in type of finite lists is defined as follows[6]:

```
data List a = Nil | a Cons (List a) | (List a) Snoc a
              where
              Nil Snoc x          => x Cons Nil
              (x Cons xs) Snoc y => x Cons (xs Snoc y)
```

The parameter `a` is a *type variable* and stands for any type[7]. Its scope is the entire right-hand-side. This declaration introduces a new type constructor, `List`, and three data con-

---

5 Recall that, in addition to connections, proper ASDL definitions can be parameterized by generic attributes of physical type.

6 The concrete syntax for ASDL lists is actually `data [a] = [] | a :< [a] | [a] :> a`

7 The following convention will be assumed in the following: type variables will always be denoted with small whereas type constructors will be denoted with capital letters.

structors[8], `Nil`, `Cons`, `Snoc`, with the following inference rules (where the premises appear above and the conclusion below the horizontal line) that specify how elements of the list type are formed:

$$\frac{}{\texttt{Nil :: List a}}\ \textit{Nil-intro}$$

$$\frac{\texttt{x :: a} \qquad \texttt{xs :: List a}}{\texttt{x Cons xs :: List a}}\ \textit{Cons-intro}$$

$$\frac{\texttt{x :: a} \qquad \texttt{xs :: List a}}{\texttt{xs Snoc x :: List a}}\ \textit{Snoc-intro}$$

These are called **introduction rules**. The following **congruence rules** are introduced as well:

$$\frac{\texttt{x :: a}}{\texttt{Nil Snoc x == x Cons Nil}}\ \textit{Nil-Snoc-intro}$$

$$\frac{\texttt{x, y :: a} \qquad \texttt{xs :: List a}}{\texttt{(x Cons xs) Snoc y == x Cons (xs Snoc y)}}\ \textit{Cons-Snoc-intro}$$

The congruence rules, which are treated semantically as introduction rules, impose additional equalities on the canonical objects, over those implied by the free type. In this particular case, we have chosen the "Cons view" of lists as the primary one because this is the one most users of applicative languages are familiar with, but it could be equally valid to choose the "Snoc view" instead. In fact, one way to understand ASDL lists is in terms of the generic patterns of induction that can be defined in terms of them: inductive definitions using Snoc lists correspond to **`for`** `i=0` **`to`** `n-1` (left-to-right) loops, whereas definitions using Cons lists correspond to **`for`** `i=n-1` **`downto`** `0` (right-to-left) loops (`n` is the size of the list). Snoc lists are more convenient at the register transfer level where, most of the times, the least significant bit is

[8] When an algebraic data type is defined by a number of alternatives greater than one, their constructors are called **sum constructors** and the patterns constructed with them can be refuted.

assumed to reside at the right end of the bus. This topic will be discussed further in the next chapter. Notice that the `Nil` constructor is shared between the two views of lists.

The semantics of algebraic data types with congruence rules are straightforward: constructors are treated as system definitions as well. The resulting system definitions can be recursive if the name of the outermost constructor on the left appears on the right of the => as well. If no congruence (pattern-matching) rule applies, the resulting expression is considered to be in normal form. For example, in the above definition of finite lists, the outermost constructor in the two congruence rules (`Snoc`) is assumed to be the name of a system definition; every occurrence of `Snoc` at the right-hand-side of any system definition is assumed to denote an application of it. This is similar to the way that Miranda's™ (version 1) lawful types are implemented.

What is the general syntax of algebraic data type declarations in ASDL? A data type `T` is defined by giving its data constructors `Ci`, where `1<i`, and stating the type of each one of them, as in the list example given above. In general, `T` can be parameterized by a number of type parameters `a1...ak` where `k>=0`. The type of the data constructors is then

```
Ci :: Ti1,...,Tin -> T a1...ak
```

where `n>=0` is the arity of the constructor and each type expression `Tij` is either `T a1...ak`, or a type expression constructed from the parameters of the type using `(,)` (cartesian product constructor), `[]` (list type constructor), `->` (function space constructor) and any other previously defined type (but not both). Notice that the above syntax guarantees that no type expression can be "negative" [94] on the defining type, i.e. the following declaration, that is usually used to make the fixpoint operator typeable in Milner's system,

```
data F a = Lambda (F a -> a)
```

is not permitted.

Algebraic data types are a very general idea and include a number of special cases, like enumerated or composite or variant types, which in other hardware description languages require separate

constructions or are missing altogether. For example, **labelled union** types (variants where two or more mutually exclusive data types are tagged with unique constructors) are notably missing from VHDL. Nevertheless, union types are instrumental in specifying bus structures since they allow different token types, i.e. instruction and data words, to coexist. In fact, data typing of tokens, as discussed in [135] eliminates the need for a control-flow graph for specifying sequential behavior. While we are still on the subject of union types, we should also mention the fact that "don't care values" can be easily specified by adjoining a nullary constructor `?type` to the constructors of `type`. Notice that `?type` is different from "undefined" (or bottom) because no pointed domains (which correspond to non terminating behavior) are allowed in ASDL.

What is the meaning of a data type in ASDL? A type `T` together with its constructors `Ci` constitutes an algebra, i.e. a set with a structure. More specifically, the meaning of a data type declaration is taken to be the **initial** `T`—algebra [88] which means that the constructors are assumed strict in all their argument positions. In chapter 4 we will see how "homomorphisms", i.e. schemes of inductive definitions, can be systematically defined on a data type, and in chapter 5 we will further generalize these schemes into attribute grammars.

Congruence rules, like the ones discussed at the beginning of this section, simply define a subset of the free data type. In many occasions in hardware design, it is necessary to define an **isomorphism** between (subsets of) two data types, i.e. abstract data types. One such occasion using list to tree isomorphisms is discussed in detail in chapter 5. Views with ephemeral data constructors [130] have been considered in the preliminary design of the consensus functional programming language Haskell as a way to reconcile data abstraction with pattern-matching. Since pattern-matching is absolutely necessary in our inductive definitions, views are natural candidates for inclusion in the design of ASDL. We use an extension of the original proposal which allows the naming of a data type and, therefore, allows the separation of the data type definitions from the definition of the isomorphism between them. As an example, we can define a data type of binary trees and

declare an isomorphism between this type and the type of finite lists aiming at revealing more divide and conquer type of parallelism[9]:

```
data Btree a = Nilt | Unit a | (Btree a) Cat (Btree a)
iso ListCat a = [a] <=> Btree a
                where
                list2cat []  => Nilt
                list2cat [x] => Unit x
                list2cat xs
                 => (list2cat as) Cat (list2cat bs)
                    where
                    (as,bs)
                     <= splitAt ((length xs) div 2) xs
                cat2list Nilt      => []
                cat2list (Unit x)  => [x]
                cat2list (x Cat y)
                 => (cat2list x) ++ (cat2list y)
```

An isomorphism declaration relates two data types with the *same* parameters by defining a pair of **coercion** functions that specify how elements of one type will be mapped to elements of the other. Notice that coercion functions can be defined by direct recursion (although inductive definitions in the above case are possible with modest syntax extensions). The isomorphism is well defined when the two coercion functions are inverses of each other. Naturally, this cannot be enforced by syntactic or type rules alone, it has to be proven for each pair of coercion functions. In appendix B we prove this fact for `cat2list` and `list2cat`. When the coercion functions are not inverses of each other, then one of the types participating in the definition of the isomorphism has to be declared as the **preferred** one. We have chosen arbitrarily, the left argument of <=> to be regarded as the preferred one. It is the responsibility of the VLSI programmer to make sure that this is actually the case, i.e. the preferred type is the representation type in the data abstraction. We will not, however, make use of asymmetrical isomorphisms in this thesis.

---

[9] The function `length` returns the length of a list, `div` is the built-in integer division operator, and `splitAt`, which splits a list in two parts at a specified position, and ++ (catenate two lists) are defined in section 4.2.

A data type can be declared isomorphic to more than one type, i.e. it can be shared. Explicit type declarations, as in the following example, are used to determine the appropriate coercions:

```
accl::((a,a)->a)->([a]=>ListCat a)->a->(ListCat a=>[a],a)
accl @ Nilt      r => (Nilt,r)
accl @ (Unit a)  r => (Unit r,a)
accl @ (x Cat y) r => (xt Cat yt,xv @ yv)
                      where
                      (xt,xv) <= rec x r
                      (yt,yv) <= rec y (r @ xv)
```

The meaning of a typing like `[a]=>ListCat a` in a type signature is that the object at the specified *input* position of type `[a]` will be coerced by applying `list2cat`. The meaning of a typing like `ListCat a=>[a]` in a type signature is that the object at the specified *output* position will be coerced to `[a]` by applying `cat2list`. It is a type error if the type inferred from the body of the definition is different from the one expected by the coercion function, i.e. different from `Btree a` in both places in the running example. The above definition, then, will be translated into the following case expression, where the necessary coercions have been explicitly inserted:

```
accl @ ts r =>
case (list2cat ts) of {
 Nilt      => (cat2list Nilt,r);
 (Unit a)  => (cat2list (Unit r),a);
 (x Cat y) => (cat2list ((list2cat xs) Cat (list2cat ys)),
               xv @ yv)
               where
               (xs,yv) <= rec (cat2list x) r
               (ys,yv) <= rec (cat2list y) (r@xv)
}
```

Notice that values of type `Btree a` appear in the text in a very restricted way, namely as a result of `list2cat` or as an argument to `cat2list`. Naturally, optimizations of the resulting case expression are possible.

A notable feature of the abstraction defined above is that it preserves the relative order of the elements in the list. This may be significant if the subsequent combinator is making use of "interfering" communication, for example when carries are involved. If the order is irrelevant

for the subsequent system combinator[10] then any "shuffling" of the input that does not duplicate or discard elements can be considered as a (total) coercion function. Butterflies, perfect shuffles, shifters, in fact any data path as the ones given in [80] can be described at this level. As an example, we define such an isomorphism between lists and trees:

```
iso Shuffle a = [a] <=> (Btree a)
                where
                list2even_odd []  => Nilt
                list2even_odd [a] => Unit a
                list2even_odd as
                 => (list2even_odd (evens as)) Cat
                    (list2even_odd (odds as))
                even_odd2list Nilt      => []
                even_odd2list (Unit a)  => [a]
                even_odd2list (x Cat y)
                 => zip2 (even_odd2list x) (even_odd2list y)

evens []           => []
evens [a]          => [a]
evens (a:<al:<as) => a:<rec as
odds []           => []
odds [a]          => [a]
odds (a:<al:<as) => al:<rec as
zip2 []      []      => []
zip2 (a:<as) (b:<bs) => (a,b):<rec as bs
```

We make use of the auxiliary system combinators `evens`, `odds` and `zip2`. Note that this time the coercions are only partial inverses (since `zip2` is undefined when the two lists are not of the same length). This can be avoided if we impose the additional constraint that the number of elements in the input list is a power of two; in this case, we can easily prove, using the lemma `for all xs, zip2 (evens xs) (odds xs) == xs`, that the coercion functions are inverses of each other and, therefore, the isomorphism is well defined.

As a closing remark, note that a type isomorphism where coercion functions are symmetrical to each other corresponds to a data type with congruence rules. We have chosen, though, to retain in ASDL the syntactic convenience that congruence rules offer in defining physical types with unit conversions.

---

[10] These are called *pointwise* combinators in chapter 4; most notable among them is the `map` operator.

## 3.5 The structural model

This is the initial, i.e. the most detailed, model of ASDL. In this model, the domain of interpretation consists of physical rather than mathematical entities. It should be clear that, as with all models, neither the structural semantic function nor its implementations is part of the language definition. The semantics are "described" (we avoid the use of the word "defined" since we are dealing with non-mathematical entities here) through a translation of kernel ASDL into a **wirelist** intermediate language with *object* (as opposed to *value*) semantics. Its abstract syntax is defined in Haskell [55] as follows:

```
type Netlist = [Comp_def]

data Comp_def
      = Def Comp_name [Generic] [Port] [Comp_instance]
type Generic = (Param_name,Type_name)
type Comp_instance = (Instance_num,Comp)
data Comp = Lib Comp_name [Attr_value] [Net]
          | Fun Comp_name [Attr_value] [Net]
data Port = In    Port_name Type_name
          | Out   Port_name Type_name
          | InOut Port_name Type_name

data Net = LVAR Net_num | OPEN
```

Care has been taken in the design of the wirelist language so that it can also accommodate future extensions of ASDL to bidirectional systems. In addition to input/output ports, components are allowed to have *generic* parameters as well. These would correspond to propagation delays, initial values of registers, contents of ROMs, and, in general, attributes of the system that are of no significance to the initial (structural) model but may be of interest to other models, i.e. behavior.

The correspondence of the kernel ASDL syntactic forms (recall that all semantic functions are applied to system definitions after elaboration) and the structural model is as follows:

1. Function application (other than application of a `join` function) is interpreted as serial input—output connection. Every occurrence of a system application is tagged with a unique

identification (corresponding to *component instantiation* in VHDL). This is what we meant by object semantics in the definition of the wirelist language above. A complaint has to be issued when *all* outputs from a component are floating, that is if the component is not contributing to the output in any way.

2. Multi-input (multi-output) systems are constructed by tupling inputs (outputs). The association of wires with ports is what the VHDL nomenclature calls *positional port association* (the other kind of VHDL port association, i.e. *by name*, is not currently available because no labelled records exist in ASDL).

3. Local clauses are interpreted as defining fanout (repetition of variables) and/or feedback (recursive local data bindings).

4. Wires joined together using a bus resolution function are identified with a common net name. That is, an expression `el` **`where`** `Cxt(x <= jfun (y,z))` where `jfun` is a bus resolution function and `Cxt` is the context of the local join definition, is translated into `(el` **`where`** `Cxt)[x/y,x/z]`. This process is repeated until no further instances of bus resolution functions are found at the right-hand-side of proper system definitions.

This completes the outline of the method used for obtaining a wirelist intermediate form expression from an ASDL kernel source. In chapter 7 we will present examples of wirelists obtained from ASDL inductive definitions.

## 3.6 Behavioral semantics

Simulating a system's expected behavior is an essential requirement of any hardware description environment. In formal system development, though, it is used in a different way than in most existing frameworks for hardware design. That is, it is not used for design verification but, rather, to reveal properties of performance that are not addressed formally (because of lack of suitable models), and to gain confidence that the specification expresses our needs (assuming

that the specification is deterministic). The last one, animating specifications, is indeed one of the strongest arguments for employing executable specifications (see, for example [50] where the difficulties of formalizing the specification are discussed). Although the meaning of this model can be given by other methods as well, for the class of unidirectional, digital systems that we examine in this thesis the most "natural" semantics is given, perhaps, by fixpoint theory [23]. In order to simplify the presentation of the semantic functions for the "common" parts of ASDL, we will consider single-output systems only; the generalization to multi-output systems is trivial, if we assume a family of projector functions.

The syntax for kernel ASDL system definitions is:

```
f (a1,...ak) => e0
                where
                v1 <= f1 e1
                  ...
                vn <= fn en
```

The following declarations are assumed for the definitions in this section:

```
a1,...,ak,v1,...,vn :: Variable (net names and generic parameters)
f,f1,...,fn :: System_definition
c :: Atom + Bus_resolution_function
e0,...,en,a1',...,ak' :: Expression
```

Since the semantics of all behavioral models of interest are fully compositional, it makes sense to parameterize the common parts by the type of the domain(s) of interpretation, say `Di`. The meaning functions for system definitions and expressions are (potentially) partial functions, defined inductively as follows (where $\lambda$—abstractions and "let" expressions have been introduced for convenience of manipulation):

```
Msys :: System_definition -> (Di^n->Di)  (n>=0 is the arity)
Msys [|c|] = Matom [|c|]
Msys [|λ(a1,...,ak) . e0 where (v1,...,vn) <= (f1 e1,...,fn en)|]
        (a1',...,an')
= let (v1',...,vn') =
      fix (λ(v1,...,vn) .
          (Msys [|f1|] (Mexp [|e1|]),...,Msys [|fn|] (Mexp [|en|])))
  in Mexp [|e0|][a1'/a1,...,ak'/ak, v1'/v1,...,vn'/vn]
Mexp :: Expression -> Di
Mexp [|f e|] = Msys [|f|] (Mexp [|e|])
Mexp [|(e1,...,en)|] = (Mexp [|e1|],...,Mexp [|en|])
```

Recursion is denoted explicitly by `fix::(Di->Di) -> Di`, the "fixpoint operator", that characterizes the least solution with respect to the partial ordering of the domain of interpretation, with the reduction rule: `fix f => f (fix f)`. We will discuss, now, the specific behavioral models of interest.

### 3.6.1 The instantaneous behavioral model

It can be seen that, by examining the right-hand-sides of `Msys` and `Mexp`, the semantics can be "fixed" by instantiating `Di` and defining `Matom` for all atoms and bus resolution functions of interest. The instantaneous (or static) behavioral model is obtained by instantiating `Di` with a flat domain, say the domain of bit values, and defining the "truth tables" (extensions) for the various gates. It should be also understood that, in case of multiple types of atomic connections (multiple **basetype** declarations), the domain of interpretation becomes a cartesian product of flat domains, one for each basetype declaration. Combinational gates can be defined in this way (where the following data declaration now belongs to the metalanguage):

```
data Bit = L | H where Matom [|c|] = c'

and2' :: Bit^2 -> Bit          xor2' :: Bit^2 -> Bit
and2' (H,H) = H                xor2' (H,H) = L
and2' (H,L) = L                xor2' (H,L) = H
and2' (L,H) = L                xor2' (L,H) = H
and2' (L,L) = L                xor2' (L,L) = L
or2' :: Bit^2 -> Bit           inv' :: Bit -> Bit
or2' (H,H) = H                 inv' H = L inv' L = H
or2' (H,L) = H                 pwr' :: Bit
or2' (L,H) = H                 pwr' = H
or2' (L,L) = L                 gnd' :: Bit
                               gnd' = L
```

The algebra defined by the flat domain of interpretation, the atoms and bus resolution functions defined on it will be called the **computational basis**. Notice that atomic systems like `pwr` with arity 0, are constant sources and play the role of global variables in a set of system definitions. Notice also that, when the bindings of local variables are recursive (systems with combinational feedback), the least fixpoint solution of the system is `bottom`, the totally undefined element. This happens because the domain of bit values is flat with a trivial order, $\texttt{bottom} \leq \texttt{H}$ and $\texttt{bottom} \leq \texttt{L}$. To avoid divergence, registers have to be introduced as atoms and the domain of interpretation has to be "lifted". The application-specific algebra associated with each instantaneous model, i.e. boolean algebra in our case, is lifted as well. This is discussed next.

## 3.6.2 The stream behavioral model

The instantaneous model describes static behavior in a domain of values `a`. An abstraction of the dynamic behavior can be obtained by changing the domain of interpretation in one of the

following ways:

- As a discrete time function `Nat->a`, parameterized by `a`, the type of the underlying flat domain of values.
- As a stream, i.e. a (potentially) infinite sequence of values.

Both of these models have the same expressive power and are interconvertible. In terms of efficiency, the discrete time model, when accompanied by memo-ization is the preferred choice for strict languages whereas lazy evaluation, as implemented in modern graph-reduction systems, is ideally suited for stream behavioral models. Our presentation will be based on the stream model, although in chapter 6 we employ the first one when defining the semantics of MOS transistors. The stream model is, in essence, equivalent to that of Kahn [70].

A stream of values of type `a` is defined as a **domain**, i.e. a set of finite and denumerably infinite sequences of elements over `a` together with an approximation ordering defined as `s1`$\leq$ `s2` if and only if `s1` is an initial segment of `s2`, i.e. the prefix ordering. Streams are asymmetrical and their heads are clearly denoted as in the following example: `<1,2,3<`. The bottom element of the domain is the empty stream `<<`. The following *continuous* operations [69] are defined on streams:

```
head :: Stream a -> a
tail :: Stream a -> Stream a
(:) :: (a,Stream a) -> Stream a
null :: Stream a -> Bool
```

Examples are:

```
head <1,2< = 1
tail <1,2< = <2<
1 : <2< = <1,2< = 1 : 2 : <<
null << = True
```

The stream (or multiplex) behavior of a combinational system (or of a bus resolution function) is obtained by "lifting" its instantaneous behavior into the stream domain, i.e. the meaning function is defined as `mpx [|c|] = lift c'`, where, in general, `c` will have multiple inputs, say `j`, and multiple outputs, say `k` (the generalization to nested products is straightforward). The definition of lifting is given in figure 3.6.1. Notice that when the system `c` is a constant source, i.e. when `j=0`, then its lifting is defined as a "cyclic" program, which makes essential use of lazy evaluation. If `j=1` (`k=1`) then, since there is no concept of a one-tuple in ASDL, lifting is as in figure 3.6.1 but omitting `zip` (`unzip`).

If we extend the notion of isomorphic types to encompass infinite data structures as well, and define a tuple of streams as isomorphic to its transposition (modulo truncation to the shortest stream), then our lifting operator can be simplified to the map functional.

The stream behavior of combinational gates is co-ordinated by registers whose behavior has to be defined from first principles in this model. A generic, i.e. polymorphic, **unit-delay** `D`, initialized by `a` of type `Di` is defined:

```
D :: Di->Stream Di
D a = λx.(a:x)
```

Sometimes, it is not convenient to keep track of the initial values. If we have other means to "calculate" the latency of a system, then we can simply ignore the output during this period and the definition of the unit-delay operator simplifies to:

```
D :: Stream Di
D = λx.(?Di:x)
```

where `?Di` is the *unknown* value of the type `Di`. This is actually the form of unit-delay used in the remainder of this thesis.

The following notes clarify two essential points with respect to synchronization and transparency of lifting:

Figure 3.6.1. Lifting of instanataneous to stream behavior

```
/* case of j>1, k>1 */
lift_{j,k} :: ((a_1,...a_j)->(b_1,...b_k))->(Stream a_1,...Stream a_j)
            ->(Stream b_1,...Stream b_k)
lift_{j,k} c' = λx.(unzip_k (map c' (zip_j x)))
map :: (a->b)->Stream a->Stream b
map = fix (λm.λf.λxs.(if null xs then <<
                                  else f (head xs):m f (tail xs)))
zip_n :: (Stream a_1,...Stream a_n)->Stream [(a_1,...a_n)]
zip_n = fix (λz.λ(x_1,...x_n).(if null x_1 then << else...
                          if null x_n then << else
                      (head x_1,...head x_n):z (tail x_1,...tail x_n)))
unzip_n :: Stream (a_1,..a_n)->(Stream a_1,...Stream a_n)
unzip_n = fix (λz.λxs.(if null xs then (<<,...<<) else
                      let (as_1,...as_n) = z (tail xs)
                          (a_1,...a_n) = head xs
                      in (a_1:as_1,...a_n:as_n)))
/* case of j=0 */
lift_{0,k} c' = unzip_k (fix (λcs.(c':cs)))
```

**Note 1:** The stream model, as defined above, does not imply anything about the synchronization mechanism used to ensure that corresponding values are brought together. Synchronization can be realized by an (implied) global clock, as in synchronous sequential circuits, in which case the unit-delay time will correspond to the clock period, and it can be implemented by a D-type

flip-flop[11]. Other types of flip-flops can be also considered as atomic in this model which is sometimes referred to as the Register Transfer Level (RTL). Synchronization can be also realized by local handshaking, even though this means that events with the same index in the streams may not actually happen at the same time. All that streams model is the relative order of events; this topic is discussed in more detail in [74] where strictness analysis techniques are employed in order to model the availability of data in a stream-based functional language similar to ours.

**Note 2:** A system `f` is called **shift-invariant** (or timeless in [62]) if it does not behave differently on account of the absolute time, i.e. for all `x`, `f (D x) == D (f x)`, where equality is taken in the stream model. Jones and Sheeran have shown in [61] that `D` itself is shift-invariant and that systems without state at all, i.e. combinational systems, necessarily have this property[12]. Therefore the lifting from instantaneous to stream behavior preserves the properties of the operators of the computational basis. A law can be proved in the (simpler) instantaneous model and then "lifted" to the stream model by lifting each of the atoms and bus resolution functions involved. The advantage of this approach is that `D` is treated uniformly along with any other system. This fact is used in section 5.4 where an algorithm for retiming of systems described in terms of temporal attribute grammars is introduced. An additional simplification is possible due to the following law, valid in the stream model only: `D (c x) == c x`, where `c` is a constant source. In words, delaying a constant signal results in the same signal (modulo some initially unknown value).

The stream behavior of composite ASDL systems, constructed using combinational systems lifted as described above, and unit-delays, naturally has to be monotonic and continuous. In the context of streams, *monotonicity* means that additional input can only result in the system producing

---

[11] To be more precise, the flip-flops that are suitable for synchronous operation must not allow at any moment direct communication from input to outputs. Master-slave or edge-triggered flip-flops have this property.

[12] It is also possible to formulate shift-invariance in category-theoretic terms by exploring the similarity between "natural transformations" (all ASDL definitions display this property) and parametric polymorphism as in [116].

additional output and *continuity* prevents any system from deciding to produce some output only after it has received an infinite amount of input.

### 3.6.3 The timing behavioral model

In the stream behavioral model, time is abstract: the unit of time corresponds to the relative difference between synchronization events. At the (detailed) **timing** level considered in this section, the reference to the time dimension is a reference to physical, i.e. *real time*. The clock signal, if present, is modelled as any other signal in the system description. At this level, flip-flops are non-atomic, i.e. can be constructed from other sub-systems, and gates have *nominal* propagation delays, i.e. each one is assigned a different delay[13].

It turns out (see Conlan Report [44]) that the detailed timing behavior can be modelled in terms of two types of delays:

- **Inertial delay**, i.e. the minimum amount of time during which a signal must persist at the input of a system in order to effect a change at the output. This type of delay is usually associated with propagation through real, physical gates.
- **Transport delay**, i.e. the "unconditional" delay of a signal. This type of delay is usually associated with wire-delay which is the dominant one in modern VLSI.

The difference between these two types of delay is illustrated in figure 3.6.2. Notice that inertial delays, effectively, filter out short duration pulses.

In the following, we present a *fully compositional model* for the timing behavior as an extension of the stream behavioral model, i.e. by keeping the same domain of interpretation (streams of values). Notice that successive values in a stream denote now the value of the signal in consecutive clock ticks, i.e. our model of time is *discrete* and *linear*. Since we want to be closer

---

[13] The stream behavioral model, in which gates are assumed to respond immediately to changes in their inputs (*ideal gates*), is sometimes referred to as *zero-delay* model. *Unit-delay* models, where each gate is assumed to have the same, non-zero delay, will not be considered here.

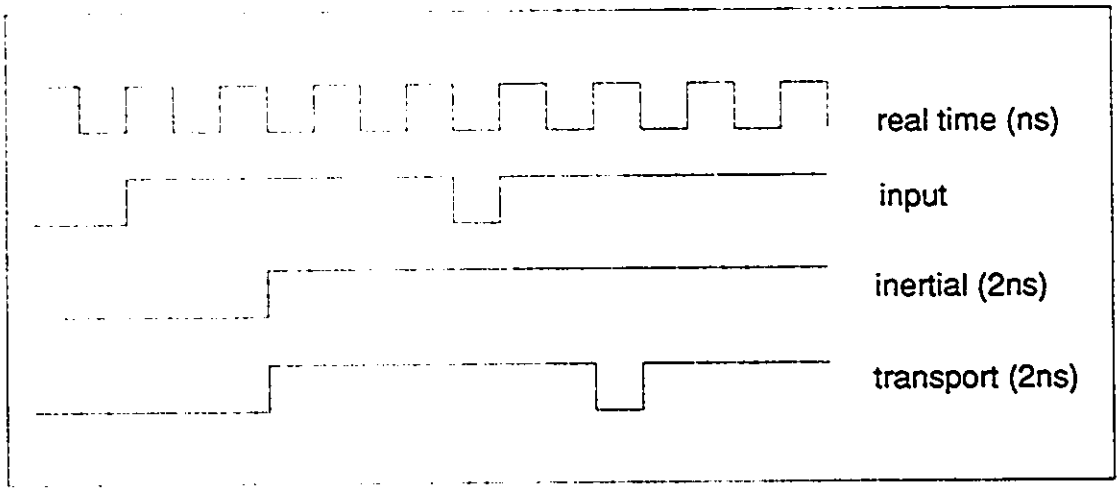



Figure 3.6.2. An example illustrating the difference between inertial and transport type delays

to the level of description offered by modern hardware description languages, each gate will be characterized by two timing parameters, the *rise time* (`tpLH`) and the *fall time* (`tpHL`). This will force us to restrict our attention to signals of a specific type, i.e. bit signals, nevertheless the generalization to more general types is straightforward. Inertial delay modelling requires the streams in *signal transition form*, i.e. signal values tagged with their duration (expressed in real time). We need two functions, one to accomplish the forward conversion, `stream2transit` and one to convert a signal in signal transition form back into a stream, `transit2stream`. The forward conversion, `stream2transit`, takes a pair of additional arguments, the first is the default initial value (associated with the specific signal type) and the second is the duration of the power-up phase. For reasons of readability, in the definition of `rowl` we make use of recursive local definitions instead of fixpoints:

```
stream2transit :: (a,Time) -> Stream a -> Stream (a,Time)
stream2transit (default,t) xs
 = let
    (ps,z) = rowl (λ(a,m).λb.(if a=b then (<<,(a,m+1))
                                  else(<(a,m)<,(b,1)))) xs (default,t)
   in concat ps ++ <z<
transit2stream :: Stream (a,Time) -> Stream a
transit2stream = concmap (λ(a,n) . (take n (fix (λas .(a:as)))))
rowl::(a->b->(c,a)) -> Stream b -> a -> (Stream c,a)
rowl f ys r = if null ys then (<<,r) else
                        letrec
                           (z,u) = f r (head ys)
                           (zs,w)= rowl f (tail ys) u
                        in (z:zs,w)
rowll::(a->b->(a,a)) -> Stream a -> Stream b
rowll f ys = let // undefined for empty streams
              (ps,z) = rowl f (tail ys) (head ys)
             in ps ++ <z<
```

The function `rowl` (and its companion, `rowll`, when no initial state is available) is a generic automaton parameterized by a transition function which produces both the output and the next state when given the current state and input[14]. In the case of `stream2transit` the state is the current value of the signal paired with a counter that measures the passage of (real) time; a "hiaton" (no-operand or <<) is produced as output when the current input is the same as the

[14] `concat` catenates a stream of streams, and `concmap f xs == concat (map f xs)`. In order to avoid proliferation of symbols, we overload the symbol ++ with catenation of two streams.

state. (note: as we will see in the next chapter, `row1` is also the name of a system combining form which can be interpreted as a "row" of connected cells, hence the name; it should be understood that the choice of names has not been undertaken with the purpose of confusing the reader, it is rather done in order to highlight the duality between implementations in "space" and implementations in "time"). The main work is now done by `idelay` which "smooths out" transitions on the signal shorter than the corresponding rise and fall times and at the same time adds a transport—type delay to the signal[15]:

```
idelay :: (Time,Time) -> Stream (Bit,Time) -> Stream (Bit,Time)
idelay (tpLH,tpHL) xs
 = let
   f (H,m) (L,n) = if n>tpLH then ((H,m+tpHL), (L,n))
                             else ((H,m), (H,n))
   f (L,m) (H,n) = if n>tpHL then ((L,m+tpLH), (L,n))
                             else ((L,m), (L,n))
   f (a,m) (b,n) = ((a,m), (b,n)), otherwise
   in row11 f xs
```

Finally, inertial (`after`) and transport-type delays are defined on bit-valued streams with `X` as their initial value (the `Bit` data type has to be modified accordingly):

---

[15] It is straightforward to modify `idelay` in order to produce the undefined value for the duration of short pulses. This defines the so-called "worst-case" model of ELLA [100].

```
data Bit = L | H | X

after :: (Time,Time) -> Stream Bit -> Stream Bit

after (tpLH,tpHL) x

 = transit2stream (idelay (tpLH,tpHL) (stream2transit (X,min tpLH

tpHL) x))

transport :: (Bit,Time) -> Stream Bit -> Stream Bit

transport (default,dt) xs = take dt ds ++ fix (λds . (default:ds))
```

Inertial delays are associated with the outputs of a system. For simplicity, we will assume that all input-output paths have the same rise and fall characteristics. The lifting from the instantaneous domain (recall that `c' = Matom [| c |]`) is similar to the one of the stream model, with the exception that each output is now post-processed with `idelay`:

```
/* when j>1,k>1 */

lift_with_inertial_{j,k} (tpLH,tpHL) c'

 = λx.(unzip_with_k (after (tpLH,tpHL)) (map c' (zip_j x)))

/* single output systems (k=1) */

lift_with_inertial_{j,1} (tpLH,tpHL) c'

 = λx.(after (tpLH,tpHL) (map c' (zip_j x)))

/* when n>1 */

unzip_with_n f = fix (λz.λxs.(if null xs then (<<,...,<<)

                              else

                              let (as_1,...,as_n) = z (tail xs)

                                  (a_1,...,a_n) = head xs

                              in (f (a_1:as_1),...,

                                  f (a_n:as_n))))
```

The same comments as in the case of the stream model are valid when j=0. Transport delays are associated with inputs and are typically back-annotated into the design description after layout has been performed and the delays for each path have been calculated from it.

The advantages of the above definition of the timing behavior compared with the traditional operational definition in terms of an "event driven-selective trace" algorithm are manyfold. Firstly, no global data structure, i.e. an "event queue", needs to be kept in order to schedule *future* events. The history of *past* values is used to compute the current output instead. A second advantage is that formal methods can be directly applied on the models in order to verify the timing behavior of a system. For example, the basic predicates "constant" and "stable" described in [53] can be easily defined on the streams in transition form using which more complex assertions, like "set-up" and "hold" times for flip-flops can be defined. Another consequence of the absence of global data structures is that there is now more potential for a distributed implementation of the simulation model. Partial evaluation could then be used to "compile" the conversions from stream to signal transition form and vice-versa.

## 3.7 Concluding comments

In this chapter, a new system description language has been described and a number of behavioral as well its structural model have been discussed. Additional models, like the Eichelberger hazard model, worst-case delay, component cost etc have been discussed in [30] for Glass, a hardware description language very similar to ASDL's kernel. Testability, e.g. SCOAP model, and fault-modelling in an applicative framework have been studied in [120]. Floorplanning in a framework similar to ours is discussed in [83]. Switch-level models are developed in chapter 6.

The major technical innovations of ASDL are inductive definitions and type isomorphisms. With respect to inductive definitions, we have to note that other hardware description languages, namely ELLA [100] and Glass [30], provide facilities for parameterized component definition based on

"applicative" syntax but these are simply macros whose semantics are not defined formally and whose termination is not guaranteed. VHDL needs additional control constructs such as "generate" blocks combined with "if" statements to achieve the same expressive power. This is not the full story though: in ASDL, every repetitive pattern can be encapsulated and freely re-used, whereas in VHDL this would require extensive use of advanced features like packages and it is not always possible since processes are not first class citizens in the language.

What is the expressive power of inductive definitions? All primitive recursive functions can be defined using them, but since ASDL is higher order, many that are not can be also defined. For example, Ackerman's function, which is known not to be primitive recursive, can be defined:

```
ack Zero k      => Succ k
ack (Succ m) k => aux (rec m) k
aux f Zero      => f (Succ Zero)
aux f (Succ n) => f (rec n)
```

Do these include *all* effective procedures (recall the definition of effective procedure from the previous chapter)? We contemplate that it would be possible to show, by a diagonalization argument similar to the one used by [81, pp. 248–250] that inductive definitions *cannot* be used to define all effective procedures. This is not actually a problem in our case since we are interested in systems, not computable functions. Our experience with inductive definitions suggests that all finite structural descriptions of interest can be defined in a straightforward way.

Regarding the second major innovation of ASDL, type isomorphisms, we note that, in the context of hardware description languages, STRICT [26] includes abstract data types with designer supplied coercions at the interface of system definitions. VHDL allows the use of explicit conversions when mapping ports in component instantiations but lacks a facility for defining generic algebraic data types. The verification-oriented system Lambda [90] approaches this problem in a different way, namely by using "pseudo-components" that perform the necessary coercions; this can be easily emulated in ASDL. Nevertheless, none of these approaches can be combined with pattern-matching and their syntax is much more verbose than ours. For an

alternative approach to isomorphisms using implicit coercions, see [124] where Milner's type inference algorithm is extended with equational unification.

In retrospective, it is clear to us now that in any practical hardware description language some form of *overloading*, i.e. "ad-hoc" polymorphism, has to be incorporated. It seems that an extension of Haskell's type classes to allow multiple parameters [32] holds the greater potential for subsuming the concepts of isomorphic types and buses with resolution used in this thesis. This is the reason why the semantics of these constructs have been defined "by example" rather than completely formally in this thesis.

Algebraic types are the most experimental feature of ASDL and serve the purpose of extending the pure structural modelling capabilities of the language into modelling aspects of behavior. The next step in the evolution of ASDL will probably be the identification of types with behavioral specifications.

In the following three chapters we introduce a framework within which system development using ASDL can be performed.

# Chapter 4 DEFINING A DESIGN STYLE IN ASDL

The pure applicative notation introduced in the previous chapter runs into some serious problems when trying to describe real world circuits. Firstly, the main mechanisms for describing circuit connectivity in ASDL *(i)* function application (which denotes serial connection), *(ii)* assembling named wires in tuples and/or lists (which denotes parallel connection) and *(iii)* using local clauses to express fanout and/or feedback, do not display those properties such as associativity or commutativity that would facilitate reasoning. This is reflected in the fact that there are only a few meaning preserving conversion rules for the structural model, namely $\alpha$—conversion and floating of local definitions. Secondly, the use of variables for naming wires fails to distinguish between local and remote communication, which is one of the distinguishing characteristics of the VLSI medium! This leads us to consider whether there is any way to enforce *design cliches* that avoid this kind of problems, therefore facilitating structural abstraction in ASDL.

We looked at the software development community for inspiration. There, a style emphasizing function-level reasoning has been advocated for many years now by Backus [5, FP], [6, FL] and more recently by Bird and Meertens [11], [3]. The essence of this style, which is heavily based on combinators[1], is that we should describe how programs, not data values, are formed. The analogy in hardware description is that we should reason about circuits and not about signals themselves. This chapter reports on the results of an investigation into the transfer of technology from the development of functional programs to circuit design.

---

[1] A combinator is a (potentially higher-order) function with no occurrences of free variables in its body.

## 4.1 Universal circuit combining forms

In this section, we will show that the expressive power of the combinatory style is equivalent to that of the applicative style [72] by defining a set of combinators using which any circuit connectivity can be described without naming the wires involved. First, we need a combinator for "serial composition". This is used frequently in this form and it is appropriate to define it in this form:

```
(f . g) x => f (g x)
```

We also need a combinator for "parallel composition". The essence of parallel composition is that the computation in any one of the components does not interfere with the computation in others[2]. Its definition in infix form is:

```
(f || g) (x,y) => (f x, g y)
```

Feedback is described by a delay-less loop:

```
loop f x => z
            where
            (w,z) <= f (x,w)
```

The reasons for choosing this apparently "non-symmetrical" definition should become clear when we present a number of laws relating these combining forms; see also [65]. Fanout is represented by a separate combinator in this style, one that replicates a wire into two copies:

```
fork2 x => (x,x)
```

We will also need a polymorphic identity combinator that passes a signal through without any modification, two more for "regrouping" tuple combinations, `rsh` (right-shift) and `lsh` (left-shift) and a "permutation" (`perm2`) combinator[3]:

```
id x    => x
```

---

[2] Parallel composition is different from μFP's [112] parallel construction, defined (using some syntactic sugar) as `[f,g] x => (f x,g x)`, which also implies fanout. We prefer to have different combinators for parallel composition and fanout.

[3] Alternatively, a "sink" combinator that forgets the wire on which it is applied has to be be defined.

```
rsh ((a,b),c)            => (a,(b,c))
lsh (a,(b,c))            => ((a,b),c)
perm2 ((a1,a2),(b1,b2)) => ((a1,b1),(a2,b2))
```

`rsh`, `lsh` and `perm2` may be regarded as unconditional signal routers. Using these circuit combinators, the full adder can be now described in signal-free form (assuming that serial composition has lower priority than parallel composition)[4]:

```
fa => or || id . lsh . id || ha . rsh . ha || id
```

The patterns `f || id` and `id || f` occur often enough to warrant their own definitions, `afst` (apply-to-first) and `asnd` (apply-to-second) respectively. Finally, serial composition is generalized to the following "beside" (`;`) operator (notice that the direction of data flow is opposite to that of serial composition):

```
(f ; g) (x,(y,z)) => ((y1,z1),x1) /* f beside g */
                     where
                     (y1,w) <= f (x,y)
                     (z1,x1)<= g (w,z)
```

A symmetrical combinator, "above" (or "below" as in Ruby [62]) could be also defined. Using the last few combinators above, the full-adder description in signal-free form can be rewritten as:

```
fa => afst or . (ha ; ha) . rsh
```

This is remarkably compact and avoids the use of the identity combinator which implies non-local communication.

If we want to enforce a synchronous design style where every feedback loop is intercepted by at-least one register, a synchronous loop combinator can be defined:

```
sloop f => loop (afst D . f)
```

where a one time-step polymorphic "unit—delay" operator (D) is assumed available (section 3.6.2). Alternatively, assuming that `sloop` is a primitive in the sequential behavioral model, the

---

[4] Compare this with the equivalent μFP-like description `[or . [sel1,sel1 . sel2],sel2 . sel2] . [sel1 .sel1,ha . [sel2 . sel1,sel2]] . [ha . sel1,sel2]` which is very difficult to be given a geometric interpretation because of the projector combinators.

delay operator can be defined in terms of it, i.e. `D => sloop id`. Mealy and Moore machines are easily expressed with the help of `sloop`, for example[5]:

```
mealy (f,state) => sloop (state || f . fork2)
```

Serial composition is associative but "beside" is not. In figure 4.1.1, we state (omitting proofs based on floating of local definitions) a number of additional laws relating the feedback operator with the other combining forms defined in this section. These rules can be used to transform functional expressions into what is called "prenex form" in Kloos's thesis [72], i.e. a form where all feedback loops have been pushed outside.

Figure 4.1.1. Fusion of feedback loops

```
f . loop g          == loop (asnd f . g)                 (L1)
loop f . g          == loop (f . afst g)                 (L2)
loop f . loop g     == loop (g ; f)                      (L3)
loop f || loop g    == loop (perm2 . f || g . perm2)     (L4)
```

There is a pleasant symmetry in the above laws which justifies the specific definition of loop that we have chosen for our presentation. These laws hold even if we substitute `sloop` for `loop` and, therefore, can be used to decompose a "larger" finite state machine description into a number of smaller ones.

A simple combinatory language like the one described above has been used for the purpose of describing FIR filter architectures [48]. The language used in [23] for the description of loosely-coupled distributed systems is also very similar. The main limitation of the set of combinators defined so far is that they only refer to pairs (more general, to cartesian products) of signals and, as a consequence, they are not adequate for expressing iterative structure as many modern architectures, e.g. systolic arrays, require. For this purpose, the full power of inductive definitions is necessary as it will be demonstrated in the next section.

---

[5] The priority of `(.)` is assumed to be lower than that of `(||)`.

## 4.2 Expressing Bird and Meertens Theories in ASDL

One way to extend the approach presented in section 4.1 to an arbitrary (but finite) number of inputs-outputs, is to make the pairing operator (,) associative. This has been demonstrated in [91] where a "box algebra" has been introduced as well. There are two problems with this approach, though: Firstly, typechecking (which reduces to a problem of cardinality checking) is undecidable and, secondly, as a consequence of the undecidability of typechecking, the applicability conditions of the box algebra laws become very intricate. Therefore, we were prompted to turn our attention to Bird and Meertens's Formalism (BMF) [12], initially applied to the theory of finite lists and then to other inductive data types. BMF theories may be used as a source of "free" laws for ASDL, in a way similar to the use of FP's algebra in $\mu$FP's [112] framework. In the literature, derivations in the BMF style have a serial interpretation whereas here we attempt to derive parallel implementations by interpreting BMF combinators as signal flow graphs. Since a data type can be declared to be isomorphic to more than one types (that is, it can have multiple representations), in the following discussion we will, in general, omit the specific mapping used and concentrate on the properties of the combinators defined on the abstract data type.

### 4.2.1 The combinators of the theory of finite lists

First, we define a set of **pointwise combinators**, i.e. combinators with non-interfering communication. `map` is the generalization of parallel composition over a homogeneous list, and `zipwith` is the generalization of `map` with binary combinators:

```
map :: (a->b) -> [a] -> [b]
map f []      => []
map f (a:<x) => (f a):<(rec x)

zipwith :: ((a,b)->c) -> [a] -> [b] -> [c]
zipwith op []       []       => []
zipwith op (a:<as) (b:<bs) => (op (a,b)):<(rec as bs)
```

Notice that zipwith is defined only when both input lists are of the same length and that, although both lists are traversed at the same rate, the induction is applied to the first one only due to the

limitations of the syntax for inductive definitions discussed in the previous chapter. Another first-order pointwise operator, `unzip2`, that generalizes `perm2` is defined:

```
unzip2 :: [(a,b)] -> ([a],[b])
unzip2 []         => ([],[])
unzip2 (a,b):< xs => (a:<as,b:<bs)
                     where
                     (as,bs) <= rec xs
```

All of these operators can be defined in the Snoc view of lists equally well because no internal dependencies exist.

In the classic formulation of the theory of lists [12], functions `inits` and `tails` are used to generate the initial and final segments of a list. We have found that two new generic combinators are more appropriate for modelling the availability of data in hardware design, `skewr` and `skewl`; both are parameterized by a unary polymorphic operator[6]:

```
skewr f []       => []
skewr f (a:<as) => a:<(map f (rec as))
```

`skewl` is defined similarly on the reverse (Snoc) view of lists.

The final pointwise operator is the square (`sq`), defined here in terms of `repeat` which applies `f` to `x` a number of `k` times:

```
repeat f 0 x     => x
repeat f (k+1) x => rec k (f x)

sq f as => repeat (map f) (length as) as
```

Next, we define some additional first-order inductive combinators, the infix catenate (++), used here with finite lists only, and `splice`, defined in terms of `splitAt` for which the identity `splitAt n xs==(take n xs,drop n xs)` holds for all `n`:

```
[]     ++ y => y
(a:<x) ++ y => a:<(rec x y)

/* splice n m xs == [xs!(n+1) .. xs!m] */
```

---

[6] When the operator involved is the unit-delay (D) we get the familiar from systolic arrays triangular skewing of data, for example, `skewr D [x0,x1,x2,x3] == [x0,D x1,D (D x2),D (D (D x3))]`.

```
splice n m xs => y2
                 where
                 (x1,x2) <= splitAt m xs
                 (y1,y2) <= splitAt n x1

splitAt 0       xs      => ([],xs)
splitAt _       []      => ([],[])
splitAt (n+1) (x:<xs) => (x:<xs1,xs2)

                         where (xs1,xs2) <= rec n x
```

We now turn our attention to combining forms with internal communication; first, we examine **directed reductions**, sometimes called fold operators in functional programming. Reduce-right (`redr`) is defined inductively on Cons lists whereas reduce-left (`redl`) is defined more naturally as an induction on Snoc lists:

```
redr :: ((a,b) -> b) -> b -> [a] -> b
redr op r []       => r
redr op r (a:<as) => op (a,rec as)

redl :: ((a,b) -> a) -> a -> [b] -> a
redl op r []       => r
redl op r (as:>a) => op (rec as,a)
```

We believe that these definitions reveal the existing symmetry between the last two operators better than existing notations.

**Accumulations** are also another common pattern of induction with binary operators. In a serial implementation, we can think of the binary operator (`op`) as a state transition function and the unit element (`r`) as the initial state of the automaton; in this case, accumulations return all the intermediate states in addition to the final state that correspond to a given sequence of inputs. The postfix operator (`scanr`) is defined as an induction on Cons lists and the prefix operator (`scanl`) is defined as an induction on Snoc lists:

```
scanr :: ((b,a) -> b) -> b -> [a] -> (b,[b])
scanr op r []       => (r,[])
scanr op r (a:<as) => (op (b,a), b:<bs)
                      where (b,bs) <= rec as

scanl :: ((a,b) -> b) -> b -> [a] -> ([b],b)
scanl op r []       => ([],r)
```

```
scanl op r (as:>a) => (bs:>b, op (a,b))
                      where
                      (bs,b) <= rec as
```

These operators are slightly different from the ones defined By Bird and Meertens for reasons that should become clear in the next chapter in which we will also discuss alternative implementations of accumulations in tree architectures.

The beside operator can be generalized to provide two operators, `rowl` (row-left) and its symmetrical, `rowr` (row-right), are also useful in hardware description. We define these operators as inductions on Cons lists:

```
rowl :: ((r,a) -> (b,r)) -> [a] -> r -> ([b],r)
rowl f [] r       => ([],r)
rowl f (a:<as) r => (b:<bs,rl)
                    where
                    (b,rx)  <= f (r,a)
                    (bs,rl) <= rec as rx

rowr :: ((a,b) -> (c,a)) -> a -> [b] -> (a,[c])
rowr f r []       => (r,[])
rowr f r (a:<as) => (rl,b:<bs)
                    where
                    (rx,bs) <= rec as
                    (b,rl)  <= f (rx,a)
```

Notice the asymmetry of the definitions above (`rowl` requires a parameter for the inductive variable whereas `rowr` does not). We will come back to this point when discussing attribute grammars. The duals of these inductive definitions, `coll` (column-left) and `colr` (column-right) which corresponds to "above", can be defined in a similar way. Notice that `scanl` is a special case of `rowl`.

By combining the combinators defined thus far, two-dimensional combinators can be defined. As an example, a rectangular `grid` combinator with connectivity on all four of its sides is defined using an auxiliary combinator uncurry:

```
grid f => rowl (uncurry (rowl f))

uncurry f (x,y) => f x y
```

Hexagonally connected arrays can be defined as grids with an extra piece of wiring as in [66]. Tree-like architectures are the subject of study of the following chapter.

As another example of two-dimensional structure description, the following definition is the ASDL implementation of the exchange sort circuit discussed in [119]:

```
xsort :: [a]  -> [a]
xsort []      => []
xsort (xs:>a) => redr insert a xs

insert (xs,a) => bs:>b
                 where
                 (bs,b) <= rowl compare a xs

/* assuming the availability of components min and max */
compare :: (a,a) -> (a,a)
compare (x,y) => (min x y, max x y)
```

The combinators defined so far are homogeneous, i.e. parameterized by a single operator. In many applications, i.e. programmable logic arrays or lattice filters, the structure is uniform but the individual cells perform different computations (albeit of the same "type"). This motivates the introduction of **heterogeneous combinators** [85]. The extension from homogeneous combinators to heterogeneous ones is straightforward. We give some examples below[7]:

```
pipeline :: [(a->a)] -> a -> a
pipeline [] x      => x
pipeline (f:<fs) x => rec fs (f x)

hmap :: [(a->b, a)]  -> [b]
hmap []           => []
hmap ((f,a):<fas) => (f a):<rec fas

hredr :: b -> [((a,b)->b, a)] -> b
hredr r []           => r
hredr r ((f,a):<fas) => f (a,rec fas)
```

---

[7] The last one (`ply`), when used in infix form, denoted by (`<-`), enables us to express pointwise combinators parameterized by operators of *any* arity in a very elegant manner. As an example application, an alternative definition of zipwith is:

```
zipwith op as bs => map (curry op) as <- bs
curry f x y => f (x,y)
```

```
hscanr :: b -> [((b,a) -> b, a)] -> (b, [b])
hscanr r []           => (r, [])
hscanr r ((f,a):<fas) => (f (b,a), b:<bs)
                         where (b,bs) <= rec fas

ply :: [a->b] -> [a] -> [b]
ply []      []      => []
ply (f:<fs) (a:<as) => (f x):<rec fs as
```

The advantage of inductive heterogeneous definitions is that the algebra of homogeneous combinators can be extended to accommodate them as well.

## 4.2.2 Homomorphisms and paramorphisms

In fact, all combinators introduced in the previous section are examples of what Bird calls "homomorphisms" (and Meertens "catamorphisms") on lists. The essence of a homomorphism on lists as defined in [10] is a structure-preserving map from the list monoid to another arbitrary monoid[8]. Homomorphisms are essential in the VLSI implementation of algorithms because they preserve *locality of reference* if the constructors of the particular data type have this property. Therefore it makes sense to devise a uniform scheme to define homomorphisms on any data type. We will start with the Cons view of finite lists which is inherently serial[9] since the constructor (:<) associates to the right.

Homomorphisms on Cons lists are specializations of the following scheme (@ is a non-curried infix operator):

```
consHom (@) r []     => r
consHom (@) r (a:<x) => a @ (rec x)
```

If the unit element is undefined, then we can define homomorphisms on non-empty lists:

```
consHom1 (@) [a]     => a
consHom1 (@) (a:<x) => a @ (rec x)
```

The archetypal example of a homomorphism on Cons lists is right-reduction. In the following

---

[8] A monoid over $a$ is an algebra $(a, @, r)$ where @ is an associative operator with unit element $r$.

[9] We use the word *serial* where others use the more semantically "overloaded" symbol *sequential*.

we redefine map, scanr and rowr as homomorphisms on Cons lists[10]:

```
redr  => consHom
redrl => consHoml
map f => consHom (@) []
         where
         a @ rest => (f a):<rest
scanr (@) r => consHom (@l) (r,[])
               where
               a @l (b,rest)) => (a @ b,b:<rest)
rowr f r => consHom (@) (r,[])
            where
            a @ (rx,bs) => (rl,b:<bs)
                           where
                           (b,rl) <= f (rx,a)
```

In fact, there is no need to restrict ourselves to binary operators only. Homomorphisms on cons lists are an instance of the more general primitive recursion scheme:

```
consElim h r []     => r
consElim h r (a:<x) => h (a,x,rec x)
```

This is similar to what Meertens calls "paramorphisms" and corresponds to the list elimination rule in constructive type theory (hence the name). We will make use of the term paramorphism in the following discussion. It comes as no surprise that homomorphisms can be expressed as a specialization of paramorphisms. What is surprising is the fact that Cons list paramorphisms can be expressed as homomorphisms, that is the follow identity holds:

```
snd . ConsHom (@) ([],r) == consElim h r
where

al @ (xl,ul) => (a:<xl,h (al,xl,ul))
```

In order to prove this equality, it suffices to show that,

```
for all x, (x,consElim h r x) == consHom (@) ([],r) x
```

where @ is defined as above. This proposition can be easily proved by simply expanding **rec** expressions on both sides of the equality symbol, not even needing an inductive proof!

---

[10] Notice that these definitions are not valid in ASDL which does not allow local combinator definitions. Nevertheless, these can be easily $\lambda$—lifted to the top level, as discussed in section 3.3.

In a similar way, we can define homomorphisms and paramorphisms over Snoc lists and binary trees. The latter are more interesting because, as has been shown by Kelly in his thesis [71], they show more potential for "divide and conquer" type of parallelism, provided that the binary tree constructor (Cat) is associative with Nilt as its (left) unit element, i.e. when the following identities hold:

```
Nilt Cat x == x (cathom1)

x Cat (y Cat z) == (x Cat y) Cat z (cathom2)
```

The following inductive function:

```
catHom :: (a->b)->((b,b)->b)->b->Btree a->b
catHom f (@) r Nilt       => r
catHom f (@) r (Unit a)   => f a
catHom f (@) r (x Cat y)  => (rec x) @ (rec y)
```

defines a unique homomorphism from binary trees parameterized by the type a to b when both equations (*catHom1*) and (*catHom2*) are satisfied. Sufficient conditions for this are:

```
catHom f (@) r (Nilt Cat x) == catHom f (@) r x (cathom1')

catHom f (@) r (x Cat (y Cat z)) == catHom f (@) r ((x Cat y) Cat z)

(cathom2')
```

This is shown below without resorting to induction, by simply expanding the **rec** expressions. We think that the fact that these properties can be shown so easily is an extra bonus of inductive definitions:

```
Left-hand-side (cathom1')

= catHom f (@) r (Nilt Cat x)

{catHom(3)}

= (catHom f (@) r Nilt) @ (catHom f (@) r x)

{catHom(1)}

= r @ x1
```

```
  where x1 = catHom f (@) r x
Right-hand-side (cathom1') = x1
Therefore, for (cathom1') to hold, 'r' must be the left unit of '@'.
Left-hand-side (cathom2')
= (catHom f (@) r x) @ ((catHom f (@) r y) @ (catHom f (@) r z))
Right-hand-side (cathom2')
= ((catHom f (@) r x) @ (catHom f (@) r y)) @ (catHom f (@) r z)
Therefore, for (cathom2') to hold, '@' must be associative.
```

As an example, the map symbol is now overloaded with a new definition which is expressed as a homomorphism on binary trees:

```
map_Btree f == catHom f1 Cat Nilt where f1 a => Unit (f a)
```

We will drop the data type subscript when the context makes it obvious. Reductions on binary trees are also expressed as homomorphisms:

```
red (@) r == catHom id (@) r
```

In a way similar to lists, paramorphisms on binary trees are defined as:

```
catElim :: (a->b) -> ((a,a,b,b)->b) -> b -> Btree a -> b
catElim f h r Nilt       => r
catElim f h r (Unit a)   => f a
catElim f h r (x Cat y) => h (x,y,rec x,rec y)
```

Every homomorphism on binary trees can be expressed as a paramorphism but, in this case, the latter has to respect the associativity and the identity element of the binary operator in order for the homomorphism to be unique. Details of the construction and the relevant consistency conditions are given in section 2.2 of [12]. The reverse construction, defining paramorphisms in terms of homomorphisms, is also possible, i.e. the following identity holds:

```
catElim f h r ==
snd . catHom f1 (@) (Nilt,r)
where
f1 (Unit a) => (Nilt,f a)
(x,rx) @ (y,ry) => (x Cat y, h (x,y,rx,ry))
```

In general, every paramorphism can be expressed in terms of a homomorphism with constructions analogous to the above. We prefer the syntactic convenience, though, that paramorphisms offer in the case of definitions that "consume their argument and keep it too".

It should be clear by now the general scheme by which homomorphisms and paramorphisms can be defined on any inductive data type. In the next section we introduce some very powerful theorems that, in a sense, serve the role of "pre-packaged" inductions for these data structures.

### 4.2.3 Fusion laws

Too many laws are as much of a problem in transformational circuit design as too few: in Luk's thesis [83], he has enumerated more than a hundred (mostly shallow) laws concerning Ruby's combining forms, and a plethora of additional laws (and ad-hoc combinators) has been introduced in subsequent publications. In a situation like this, the number of options facing a designer is overwhelming. Ideally, only a single law should be applicable at every step. In this section we introduce such a law for each data type. These laws permit the *fusion* of the composition of a function and a paramorphism (homomorphism) into a single paramorphism (homomorphism). They are known with other names in the literature, promotion theorems in BMF, function combination and tupling strategies in [104]. The versions presented here are proved for the instantaneous behavioral model but they can be promoted to sequential behavior using the lifting method discussed in section 3.6.2.

The fusion law on cons paramorphisms is stated and established first:

**Proposition** *(consElim fusion)*:

```
g . consElim h e == consElim k (g e)
```

when for all x,y,z, g (h (x,y,z)) == k (x,y,g z)

**Proof**: By structural induction on Cons lists.

Base case

left-hand-side

{instantiate with []}

= g (consElim h e [])

{consElim(1)}

= g e

right-hand-side

= consElim k (g e) []

{consElim(1)}

= g e

Therefore, left-hand-side == right-hand-side

Inductive assumption

g (consElim h e x) == consElim k (g e) x

Inductive step

left-hand-side

{instantiate with a:<x}

= g (consElim h e a:<x)

{consElim(2), expand rec}

= g (h (a,x,consElim h e x))

```
{assumption}
= k (a,x,g (consElim h e x))
{Inductive assumption}
k (a,x,consElim k (g e) x)
{fold consElim(2)}
= consElim k (g e) a:<x
Therefore, left-hand-side == right-hand-side
```

Q.E.D.

The following proposition is a corollary of the *consElim fusion* law:

**Proposition** *(consHom fusion)*:

```
g . consHom (@) r == consHom ($) (g r)
when for all x,y,  g (x @ y) == x $ (g y)
```

As an example application, we prove the following law:

**Lemma** *(map-scanr)*:

```
(q || map q) . scanr (@) e == scanr ($) (q e)
when for all a,b, q (a @ b) == a $ (q b)
```

**Proof:**

```
Assuming the following definitions:
f => q || map q
a @1 (b,bs) => (a @ b,bs)
a $1 (b,bs) => (a $ b,bs)
and the definition of 'scanr' as a cons homomorphism, the lemma to be
proven becomes equivalent to:
```

```
f . consHom @1 (e,[]) == consHom $1 (q e,[])
```

which, according to *consHom fusion* is true when:

```
f (a @1 (b,bs)) == a $1 (f (b,bs))
```

We will prove that this condition holds:

```
Left-hand-side
{definitions of @1,f}
= (q || map q) (a @ b,bs)
{definition of (||)}
= (q (a @ b),map q bs)
{assumption}
= (a $ q b,map q bs)
{definitions of $1,f}
= a $1 f (b,bs)
= Right-hand-side Q.E.D
```

There are symmetrical laws for Snoc lists. The proof of other BMF laws, such as the distribution of map over composition or Horner's rule, can be also performed in an equational style using the fusion laws.

We now turn our attention to the fusion law for binary tree paramorphisms:

**Proposition** *(catElim fusion)*:

```
g . catElim f h e == catElim (g . f) k (g e)
when  g (h (x,y,x1,y1)) == k (x,y,g x1,g x2)
```

**Proof:** By structural induction on binary trees.

Case of Nilt:

```
Left-hand-side
```

{instantiate with Nilt}

= g (catElim f h e Nilt)

{catElim(1)}

= g e

{fold catElim(1)}

= catElim (g . f) k (g e)

= Right-hand-side

Case of Unit:

Left-hand-side

{instantiate with (Unit a)}

= g (catElim f h e (Unit a))

{catElim(2)}

= g (f a)

= (g . f) a

{fold catElim(2)}

= catElim (g . f) k e (Unit a)

= Right-hand-side

Case of Cat (Inductive assumptions):

g (catElim f h e x) == catElim (g . f) k (g e) x

g (catElim f h e y) == catElim (g . f) k (g e) y

Case of Cat (Inductive step):

Left-hand-side

= g (catElim f h e (x Cat y))

{catElim(3)}

```
= g (h (x,y,catElim f h e x,catElim f h e y))
{assumption}
= k (x,y,g (catElim f h e x),g (catElim f h e y))
{Inductive assumptions}
= k (x,y,catElim (g . f) k (g e) x,catElim (g .f) k (g e) y)
{fold catElim(3)}
= catElim (g . f) k (g e) (x Cat y)
= Right-hand-side Q.E.D.
```

The fusion law for homomorphisms on binary trees is a corollary of *catElim fusion* proved above:

**Proposition** *(catHom fusion)*:

```
g . catHom f (@) e == catHom (g . f) ($) (g e)
when for all x,y, g (x @ y) == (g x) $ (g y)
```

The general pattern of fusion laws is now obvious. Their generalization to other inductive data types is similar to the one discussed in [87].

## 4.3 Concluding comments

In this chapter we have investigated the use of ASDL in defining structural abstractions. Others [97] have advocated the advantages of enforcing structural constraints for the purpose of facilitating formal verification. The constructs suggested in their work, "wire-in-series" and "wire-in-para", correspond to our "beside" and "para" and we have demonstrated how even more advanced abstractions can be defined in our language, therefore making the goal of "Design For Verifiability" (DFV) more tangible. In particular, our fusion laws can be perceived as canned induction proofs, greatly facilitating the task of formal verification. Anyone who had experience

reasoning with array indices in traditional systolic array design methodologies [75] should be able to appreciate the point in this argument.

We have found that higher order inductive definitions were instrumental in encapsulating commonly used patterns of data-independent recursion, such as homomorphisms and paramorphisms, associated with each inductive data type. The "form" of each data type, in general, dictates the "structure" of the resulting circuits. The set of atoms has to be extended with primitive agents non-synchronizable by a single clock (such as serial-in-parallel-out or parallel-in-serial-out shift registers) if we want to be able to describe mixed data and time abstractions, for example bit-serial architectures [65]. This should be accompanied by laws expressing various scheduling and resource allocation alternatives.

One additional advantage of the combinatory style is that it can be given an intuitive geometric interpretation (relative placement and global routing) as well. This is exploited in the work of [9] where a "Calculus of Nets" for systematic hierarchical design is described. Many different topologies correspond to the same structural description and, in this sense, the combinatory style allows floorplanning concerns to be dealt with early in the design phase.

One final comment concerns the use of the laws presented in this chapter in a silicon compilation system. A practical system should employ some kind of cost measurement. Blelloch has shown in his thesis [14] that a pair of measures, the vector count (total number of operations) and the step count (length of the critical path of the computation), is the most flexible way of measuring cost in loosely coupled architectures and Skillicorn et.al. [121] have calculated these cost measures for the basic BMF combinators. Their "cost calculus" could be extended to the new combinators introduced in this chapter relatively easily. Recent work on "algorithmic skeletons" [36] could be also used in devising the appropriate metrics. A note of caution is due here, though: non-trivial derivations do not proceed by monotonically reducing cost. In our personal experience we believe that a more modest approach, *guided synthesis*, similar to the system built by Vogt et.al. [128]

using the Cornell synthesizer generator [106], where the designer interactively selects the rule to be applied next, is the most promising one for the near future.

# Chapter 5 TEMPORAL ATTRIBUTE GRAMMARS

One characteristic of the circuits derived using the fusion laws of the previous chapter is that dataflow is uni-directional. This is a mere consequence of the fact that homomorphisms and paramorphisms compute attributes of the entire data structure in terms of attributes of its subparts, i.e. forward attributes only. This has prompted many researchers, most notably M. Sheeran and G. Jones [62] to abandon the applicative framework in favor of the more expressive (binary) relational notation which abstracts away from the direction of dataflow. This, in turn, introduces many problems related to realizability and animation of relational descriptions. In this chapter we investigate the applicability of the attribute grammar paradigm in hardware description in view of the aforementioned problems.

Attribute grammars [51] have initially been proposed as a method for specifying semantics of programming languages, but nowadays are mainly used as a tool to specify syntax-directed translations in compiler construction. Language definitions in attribute grammar notation tend to be more procedural than pure denotational semantics ones, but, at the same time, are modular enough to allow powerful abstractions to be defined. Inherited attributes can be intuitively interpreted as counter dataflow, and many related concerns, like deadlock detection, can be restated in terms of attribute grammars. In the following sections we are going to formulate these concepts as an extension of the BMF theory presented in chapter 4.

## 5.1 The parallel prefix operator

We will motivate our presentation with a very familiar example, the prefix operator, or left accumulation. This higher-order operator is at the heart of many parallel algorithms for computational

geometry, searching and sorting, pattern-matching and matrix computations among others [76]. Assuming that e is a left-unit of @, its definition as a homomorphism on binary trees is[1]:

```
scanl @ e
 => catHom f1 (@1) (Nilt,e)
    where
    f1 a => (Unit e,a) // because e @ a == a
    (x_tree,x_val) @1 (y_tree,y_val)

     => (x_tree Cat (map (@ x_val) y_tree),x_val @ y_val)
```

This is a hopelessly ineffective definition in terms of space usage! A more effective definition, which will be given the name "parallel prefix", can be derived when @ is also left-associative, in addition of having e as its left-unit:

```
scanlt @ e t  => scanlti @ e t e

scanlti @ e Nilt r       => (Nilt,e)
scanlti @ e (Unit a) r   => (Unit r,a)
scanlti @ e (x Cat y) r => (xt Cat yt,xv @ yv)
                           where
                           (xt,xv) <= rec x r

                           (yt,yv) <= rec y (r @ xv)
```

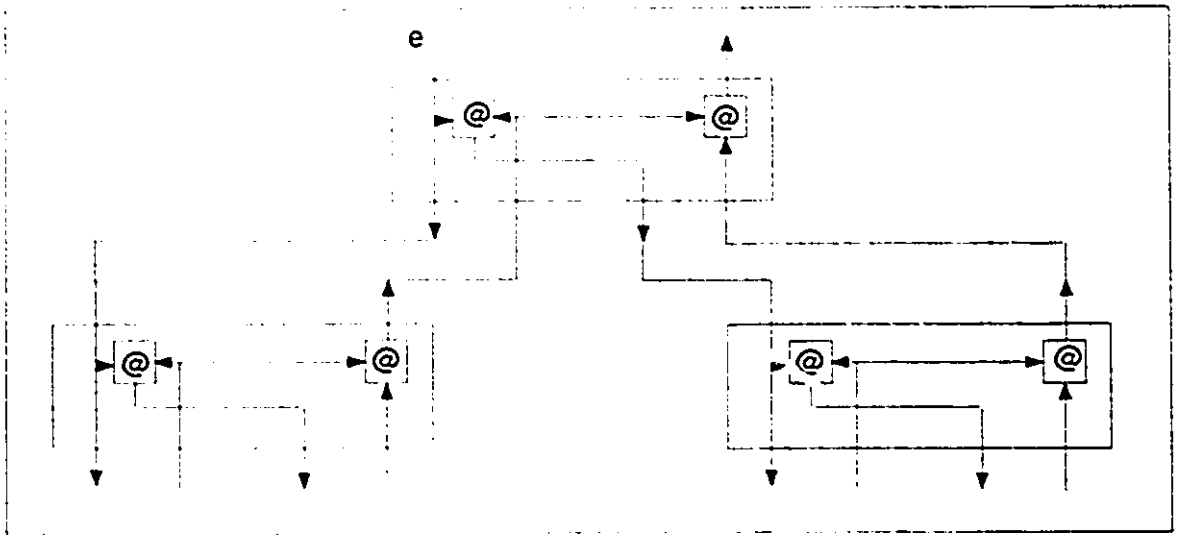
Assuming that @ is a constant time operator, this is a $O(log_2 n)$ time complexity definition that makes use of bidirectional dataflow. It cannot be defined as a homomorphism! Its schematic interpretation (in case of four inputs) is given in figure 5.1.1. The interested reader should compare this structure which has a *H* layout topology with the highly irregular structure proposed by [47] for the parallel prefix.

The question that we will try to answer in the following sections is whether a rigorous procedure to transform circuits like `scanl` into `scanlt` exists, without resorting to descriptive methods, like the up and down sweeps commonly found in the literature [14] which attempt to convince us that the transformation works correctly operationally. It turns out that in order to be able to

---

[1] In case that the operator @ does not have a (left) unit, some additional work is needed, i.e. the prefix operator is `asnd (@ r) . scanl @ r`

Figure 5.1.1. An instance of the implementation of *scanIt* when used in conjunction with the *ListCat* isomorphism




do so, we need to extend homomorphisms into inductive definitions parameterized by the *context* within which their elaboration takes place.

## 5.2 Definitions and background

In order to motivate the following discussion, we will first attempt to re-interpret the definitions of the prefix operator in terms of attribute grammars. The basic intuition is to associate inductive data types with context-free grammars (CFGs), and each term of the algebra generated by the data type definition with a derivation tree. These derivation trees are now independent of any parsing algorithm. Traditionally, each nonterminal in the derivation tree is decorated with two disjoint (and possibly empty) set of *attributes* which describe its meaning. An attribute which directly depends on attributes of children nodes is called **synthesized** and is normally visualized as going upwards in the derivation tree. Conversely, an attribute which depends on attributes of parent and/or sibling nodes is called an **inherited** attribute and is normally visualized as going downwards in the derivation tree. In order to make an attribute grammar from a CFG, a set of attribute evaluation rules is associated with each production of the grammar.

Traditionally, attribute grammars are defined as translations from a *syntactic sort* (abstract syntax trees) to a *semantic sort*, i.e. the type (meaning) of the attribute grammar. We extend this definition by considering attributes of non-terminals as sons of the corresponding non-terminal and by identifying terminal symbols with (synthesized) attributes of a semantic sort, i.e. their value domain. Then, an attribute grammar is a mapping $\alpha \rightarrow \beta$ where $\alpha$ and $\beta$ are semantic sorts. More formally:

**Definition:** A Temporal Attribute Grammar (TAG) consists of:

1. An underlying set of (possibly mutually inductive) algebraic data types $T_i$ parameterized by a set of types $t_k$. One of the $T_i$s is designated as the *start symbol* of the TAG.
2. Each $T_i$ has two disjoint sets, one of synthesized attributes and one of inherited attributes, associated with it. The start symbol has no inherited attributes associated with it and each $t_k$ has a single synthesized attribute associated with it.
3. With each variant of each data type in the grammar we associate a set of *attribution rules* of the form $a_0(T_{i1}) = f(a1(T_{i2}),....,ak(T_{ik}))$ where $T_{im}$ are either $T_i$s or $t_i$s and $a_{im}$ their attributes. The function $f$ has sort $(v1,...vk) \rightarrow v0$ where $vi$ is the sort of attribute $a_i$. These rules *define* all and only the synthesized attributes of the symbol on the left-hand-side of the data type definition, and the inherited attributes of the symbols on the right.

Notice that, in this definition, it is not assumed that the resulting attribute grammar is *normalized* [41], i.e. the evaluation rules are allowed to use any attribute and not only attributes defined outside the specific production (the inherited attributes of the left-hand-side symbol and the synthesized attributes of the right-hand-side symbols). Also, no restriction is placed upon the sort of attributes; this can be any ASDL type, including functional ones.

As a first example, we reconsider the definition of `scanl` given in the previous section as a TAG with two synthesized attributes, *val* of sort $a$, which computes partial reductions, and *tree* of sort *Btree a*, which computes the first part of the result. Now, in order to make the connection more obvious, the standard notation used by the attribute grammar community is employed in the following definition ($e$ is assumed a constant):

$$
\begin{aligned}
T &:= Nilt && \begin{cases} T|tree &= Nilt \\ T|val &= e \end{cases} \\
T &:= (Unit\ a) && \begin{cases} T|tree &= Unit\ e \\ T|val &= a \end{cases} \\
T &:= (x\ Cat\ y) && \begin{cases} T|tree &= x|tree\ Cat\ (map\ (\oplus\ x|val)\ y|tree) \\ T|val &= x|val \oplus y|val \end{cases}
\end{aligned}
$$

Figure 5.2.1. An attribute grammar description of *scanl*

The value at the root (start symbol) is taken as the pair of attributes *(tree,val)*. Notice that in this definition, which uses synthesized attributes only, a part of the syntax tree itself is grafted into the current tree resulting into what other authors call "Higher-order Attribute Grammars" [42]. This gives us a direction for optimization: introduce inherited attributes with the objective of reducing the order of the TAG. When the TAG becomes first order, the form of the TAG will be optimal because it will follow the form of the input data structure. Attribute grammars also offer insight about the means with which to achieve this objective: *factorize the higher order functionality into functional dependencies between synthesized and inherited attributes.* In our case, we introduce an inherited attribute *rep*, which broadcasts towards the left subtree the already computed reduction of the right subtree in order to avoid recomputing it each time:

$$
\begin{aligned}
T &:= root && \begin{cases} T\lfloor tree &= root\lfloor tree \\ T\lfloor val &= root\lfloor val \\ root\lfloor rep &= e \end{cases} \\
T &:= Nilt && \begin{cases} T\lfloor tree &= Nilt \\ T\lfloor val &= e \end{cases} \\
T &:= (Unit\ a) && \begin{cases} T\lfloor tree &= Unit\ T\lfloor rep \\ T\lfloor val &= a \end{cases} \\
T &:= (x\ Cat\ y) && \begin{cases} T\lfloor tree &= x\lfloor tree\ Cat\ y\lfloor tree \\ T\lfloor val &= x\lfloor val\ @\ y\lfloor val \\ x\lfloor rep &= T\lfloor rep \\ y\lfloor rep &= T\lfloor rep\ @\ x\lfloor val \end{cases}
\end{aligned}
$$

Figure 5.2.2. An attribute grammar description of *scanlt*

The translation between inductive definitions and the traditional attribute grammar notational must be obvious by now: *attribute grammars are higher-order homomorphisms* [92], i.e. homomorphisms parameterized by inherited attributes. One wonders if a generic scheme, that captures all possible TAGs on binary trees, similar to the case of paramorphisms and paramorphisms defined in the previous chapter, exists. It turns out that their definition is straightforward (up and dn are regarded as infix operators for readability reasons):

```
catAG f up r dn Nilt ti      => r ti
catAG f up r dn (Unit a) ti => f a ti
catAG f up r dn (x Cat y) ti
 => (xs,ys) up ti
    where
    xs <= rec x xi
    ys <= rec y yi
    (xi,yi) <= (xs,ys) dn ti
```

Homomorphisms, as defined in the previous chapter, are special cases of attribute grammars where inherited attributes play no role. Specifically, if the underscore character denotes a "dummy" variable (an empty context), then we can easily show that the following equivalence is true:

**Proposition (5.2.1):** For all binary trees t,

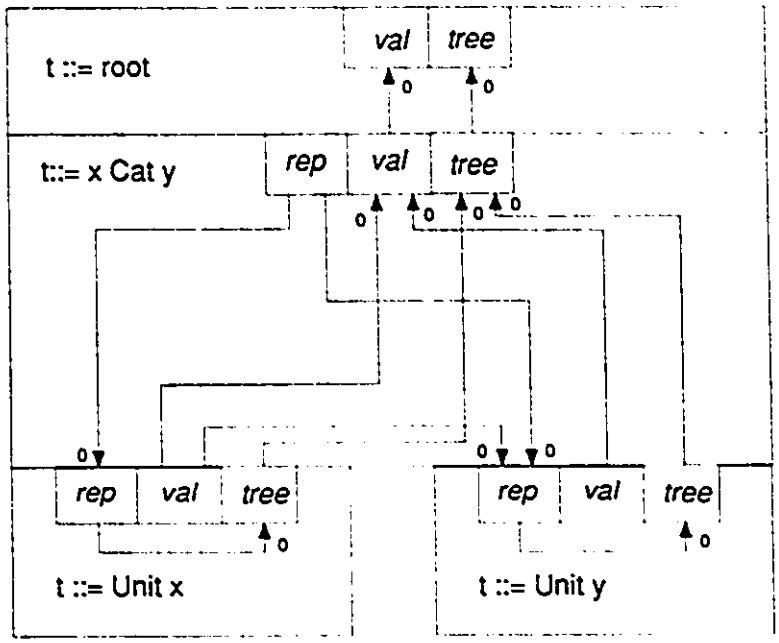
```
catHom f @ r t == catAG f1 up r1 dn t _
                  where
                  f1 a ti => f a
                  (x,y) up ti => x @ y
                  r1 ti => r
                  (x,y) dn ti => (ti,ti)
```

In any notation that allows bidirectional data flow, the question of deadlock detection naturally arises. Before we can formulate a criterion for the static detection of well-formed TAGs, we need some auxiliary definitions. The attribution rules of each variant of a data type induce a weighted directed graph called the **local dependency graph** (*LG*). A triple *(a1,a2,w)* belongs to *LG* if *a1* is an attribute instance that depends on attribute instance *a2* in the set of attribution rules and *w* is the number of delay units between them. Since we have *not* restricted ourselves to normalized attribute grammars only, the local dependency graph can contain feedback loops. By pasting together all the local dependency graphs a **compound dependency graph** (*G*) of the TAG is defined.

**Definition (5.2.1):** A Temporal Attribute Grammar is **well defined** if its compound dependency graph contain no cycles of zero weight.

Circularity detection is known to be of exponential complexity on the size of the grammar. The addition of weight computation does not, in our opinion add to the complexity of the algorithm which should perform reasonably well for typical hardware description cases. A similar test has been implemented in the dataflow language Lucid [129]. The dependency graph of the purely

Figure 5.2.3. Compound dependency graph of the *scanlt* TAG




combinational system defined by `scanlt` is shown in figure 5.2.3 (omitting the trivial case of `Nilt`). It contains no zero-weight cycles, therefore `scanlt` is well defined.

## 5.3 Coupling Temporal Attribute Grammars

In this section we explore the possibility of expressing the composition of temporal attribute grammar descriptions in the same spirit as the fusion laws of the previous chapter. This will remove a formidable obstacle on the part of the circuit designer, that of identifying the inherited attribute(s) that are propagated away from the host broadcasting information towards the leaf nodes. The process of synthesizing an attribute grammar containing both synthesized and inherited attributes from one with synthesized attributes only does not have a general solution. It depends very much on the properties of the operators involved and on our ability to invent the appropriate inherited attributes that "summarize" the information to be broadcasted in a compact form. By compact we mean that if we want the attribute grammar description to be useful as a circuit description, the attributes have to be zero-order objects. This is very similar to the situation with continuation-based transformations [58].

It is an unfortunate consequence of the tight coupling between inherited and synthesized attributes in an application that no universal fusion laws for attribute grammars have been discovered. The situation is far from hopeless, though. First, we make the observation that pointwise operators like `map` or `zipwith` do not offer any possibility for introducing inherited attributes because their essence is that they operate in a non-interfering parallel mode. Among the remaining operators, accumulators offer the most potential for improvement because of the large number of intermediate results. We have found that the following theorem captures the vast majority of the cases where inherited attributes are introduced as a result of composing two homomorphisms on binary trees:

**Theorem 5.3.1** For all binary trees `t`,

```
(afst (map k) . scanl @ e) t
== catAG f up r dn t (k e)
   where
   f a ti => (Unit ti,a)
   ((xt,xv),(yt,yv)) up ti => (xt Cat yt,xv @ yv)
   r ti => (Nilt,e)
   ((xt,xv),(yt,yv)) dn ti => (ti,ti $ xv)
```

when `@` is left-associative with left-unit `e` and `k (a @ b) == (k a) $ b`

The transformation of `scanl` to `scanlt` is a corollary of the above theorem. The proofs of theorem 5.3.1 and of the corollary can be found in appendix C.

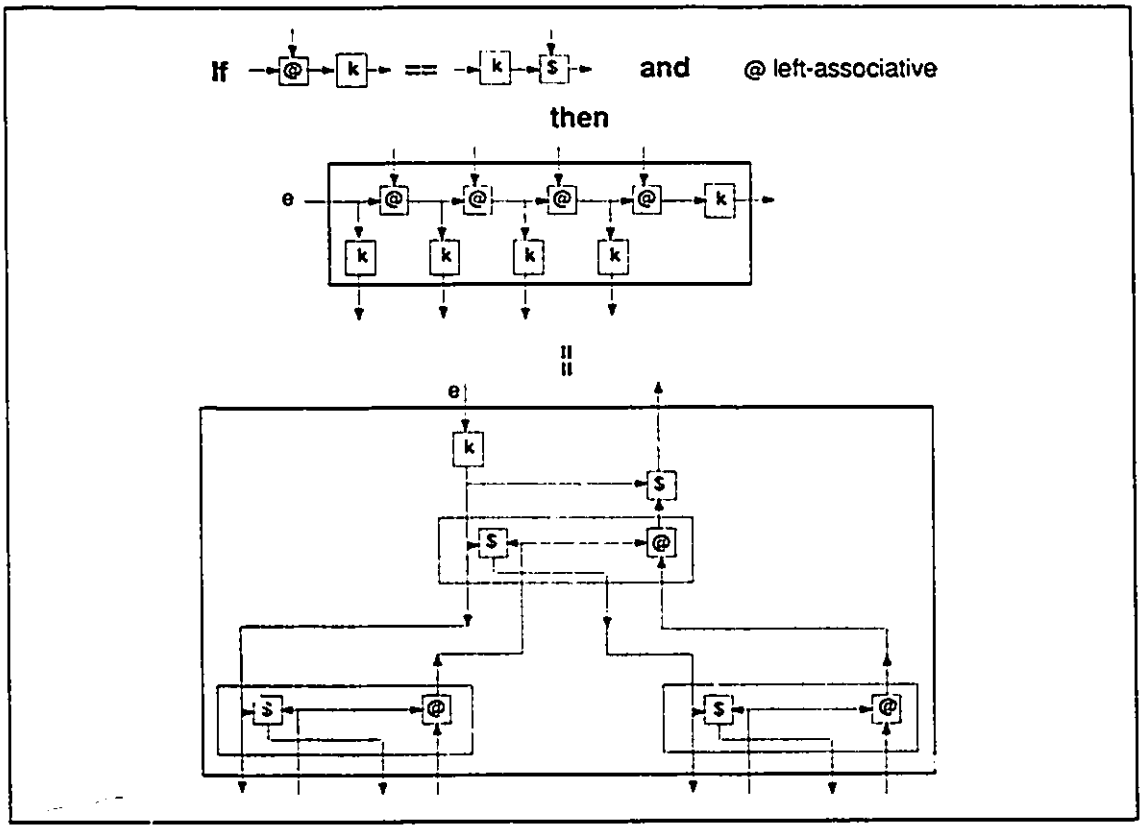
The above transformation is possible even when no identity element exists; the root case has to be modified accordingly:

**Theorem 5.3.2** For all binary trees `t`,

```
(map k || k . scanl @ e) t
== (bt, e0 $ u)
   where
   e0 <= k e
   (bt,u) <= catAG f up r dn t e0
   f a ti => (Unit ti,a)
   ((xt,xv),(yt,yv)) up ti => (xt Cat yt,xv @ yv)
   r ti => (Nilt,e)
   ((xt,xv),(yt,yv)) dn ti => (ti,ti $ xv)
```

when `@` is left-associative and `k (a @ b) == (k a) $ b`

Figure 5.3.1. An instance of Theorem 5.3.2




There exist, naturally, dual theorems for right accumulations. As an application of this we now

consider the transformation of a radix-2, ripple-carry adder[2] to a tree of carry-lookahead adders. We don't have any claims of innovation here, this is a design that has been studied thoroughly in the literature for the last 50 years. What we claim, though, is that our methodology offers novel insights about the rationale of this transformation and provides useful directions for similar situations where rippling logic is involved.

The definition of the ripple-carry adder from page 16 (most significant bit first) can be rewritten as[3]:

```
adder (abs,cin) => rowr fa cin abs
fa (c,(a,b)) => (s,co)
                where
                s  <= fsum (c,(p,g))
                co <= fcarry (c,(p,g))
                (p,g) <= ha (a,b)
fsum (c,(p,g))   => xor2 (p,c)
fcarry (c,(p,g)) => g \/ (p & c)
ha (a,b) => (p,g)
            where
            p <= xor2 (a,b) // carry propagate
            g <= a & b      // carry generate
```

Using *map—rowr* fusion and the straightforward equality:

```
rowr f r == (y,zipwith ($) xs ys)

where (y,ys) <= scanr (@) r and f (a,b) => (a $ b,a @ b)
```

we get:

```
adder (abs,cin) => (cout,res)
                   where
                   (cout,cs) <= scanr fcarry cin pgs
                   pgs <= map ha abs
                   res <= zipwith fsum pgs cs
```

The local definition `res <= zipwith fsum pgs cs` above is an instance of the scheme[4]

---

[2] You should consult [73] for an excellent presentation of the derivation of the binary ripple-carry adder from the specification of unsigned integer addition using the fold-unfold methodology.

[3] In this section, we use the more mnemonic infix operators `&` and `\/` in place of `and2` and `or2`, respectively.

[4] `S` is a combinator from combinatory logic whose reduction rule is `S f g x => f x (g x)`.

`S f g x` with `f x = zipwith fsum x, g x = fst (scanr fcarry cin x)` and `x = pgs`. Bird [11] has studied this kind of schemes using the fold-unfold methodology with the purpose of eliminating multiple traversals of the input data structure. We will try to bring the adder's description in a form where the applicability conditions for the dual form of theorem 5.3.2 are satisfied. Unfortunately, `fcarry` is not an associative operator. We therefore define an operator `@` as follows:

```
(p1,g1) @ (p2,g2) => (p1 & p2, (g1 & p2) \/ g2)
```

and prove that this operator is associative.

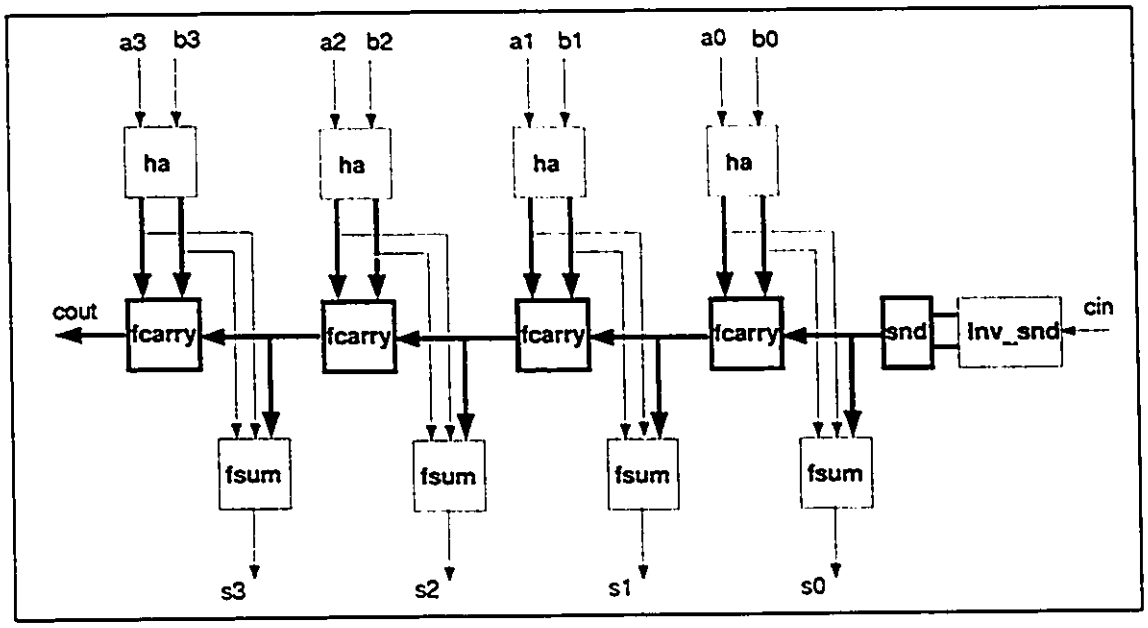
**Proof of associativity of @:**

```
We have to prove that:
((p1,g1) @ (p2,g2)) @ (p3,g3) == (p1,g1) @ ((p2,g2) @ (p3,g3))
Left-hand-side
= (p1 & p2, (g1 & p2) \/ g2) @ (p3,g3)
{...}
= ((p1 & p2) & p3, (((g1 & p2) \/ g2) & p3) \/ g3)
{associativity of & and \/, & distributes over \/}
=(p1 & p2 & p3, (g1 & p2 & p3) \/ (g2 & p3) \/ g3)
{associativity of & and \/}
=(p1 & (p2 & p3), (g1 & (p2 & p3)) \/ ((g2 & p3) \/ g3))
{definition of operator @}
= (p1,g1) @ (p2 & p3, (g2 & p3) \/ g3)
= Right-hand-side Q.E.D.
```

The conditions for the application of the *map-scanr* lemma will be satisfied if we "invent" another combinator `k` with the property that `k (a @ b) == fcarry (k a, b)`. It turns out that the combinator `snd` satisfies this property because:

```
snd ((p1,g1) @ (p2,g2))
= snd (p1 & p2, (g1 & p2) \/ g2)
= (g1 & p2) \/ g2
{definition of fcarry}
= fcarry (snd (p1,g1), (p2,g2))
```
Q.E.D.

Figure 5.3.2. Another view of a ripple carry adder




Assume, now, that a system `inv_snd` with the property `snd (inv_snd x) == x` exists[5]. An instance of the resulting ripple carry adder is depicted in figure 5.3.2. The *map-scanr* lemma can then be applied on the highlighted section of this schematic deriving the following description:

```
adder (abs,cin)
 => (cout,sout)
    where
    (cout,cs)
      <= (snd || map snd . scanr (@) (inv_snd cin)) pgs
    pgs <= map ha abs
    sout <= zipwith fsum pgs cs
```

[5] This could be the pseudo-component that pairs the input with the unkown value of the bit type, i.e. `inv_snd x => (?Bit,x)`. Our strategy will be to "cancel" this pseudo-component by circuit transformation.

Now, all the conditions of Theorem 5.3.2 are satisfied. The result, after factoring in the pointwise definitions for `pgs` and `res` which only affect the leaf nodes, is the following TAG expressed in ASDL as:

```
adder (abs,cin)
 => (cout,out)
    where
    cout <= fcarry (cin,(p0,g0))
    ((p0,g0),out)
      <= catAG afa up dn nilt abs cin
         where
         nilt ci => (ci,Nilt)
         ((xt,xv),(yt,yv)) up ci
            => (xv @ yv, xt Cat yt)
         ((xt,xv),(yt,yv)) dn ci
            => (fcarry (ci,yv), ci)
         afa (a,b) ci
           => ((p,g), Unit (fsum (ci,(p,g))))
              where
              (p,g) <= ha (a,b)
```

It is the packaging of up and dn in a single circuit abstraction that produces the well-known carry-lookahead generator. This case study shows that there is no "intrinsic" quality in binary adders (other than associativity of addition) that allows us to perform the transformation and therefore this principle is applicable in similar circuits as well.

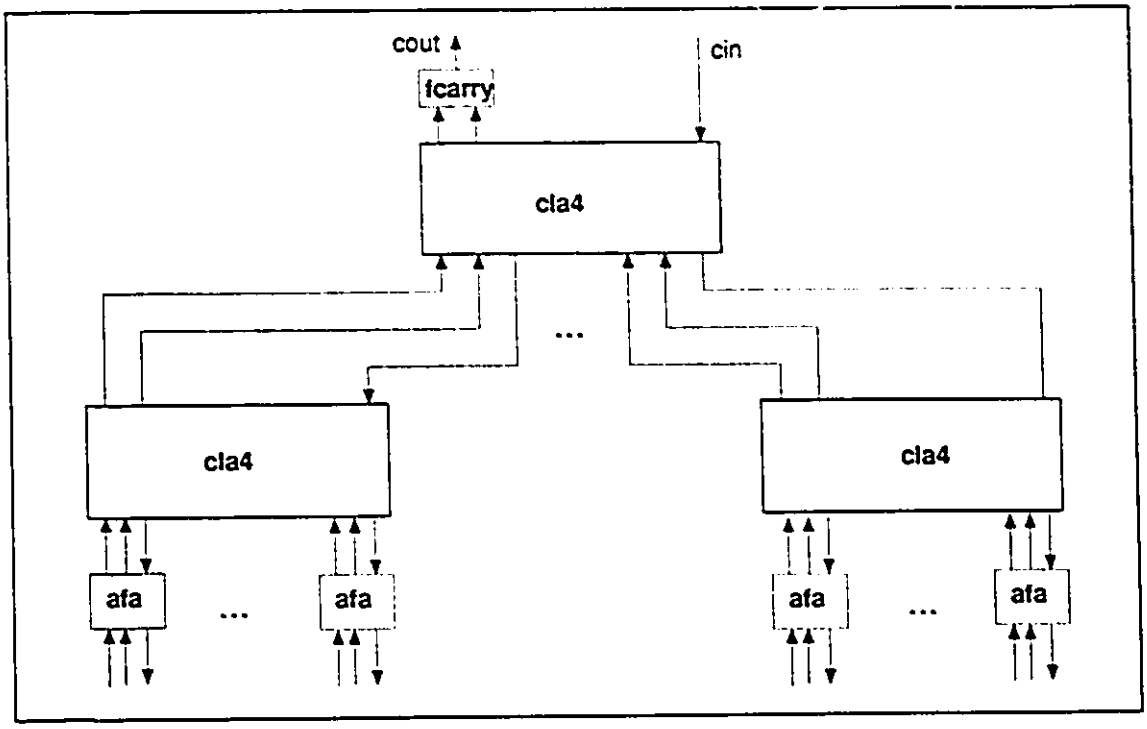
The breadth of the resulting trees can be increased (it can even be made different from level to level) if we choose general instead of binary trees:

```
data Gtree a = Node [Gtree a] | Leaf a
```

The TAG needed to describe our carry-lookahead adder on this data type makes use of `scanr` and `redr1` defined on cons lists:

```
adder (ab,cin)
 => (cout,out) // root
    where
    cout <= fcarry (cin,(p0,g0))
    ((p0,g0),out) <= cla_adder1 ab cin
```

Figure 5.3.3. An instance of the carry-lookahead adder




```
cla_adder1 (Leaf (a,b)) t_rep
 => afa (a,b) t_rep
    where
    afa (a,b) ci
      => ((p,g), Unit (fsum (ci,(p,g))))
         where
         (p,g) <= ha (a,b)
cla_adder1 (Node ts) t_rep
 => (t_val,Node ts_tree)
    where
    (ts_val,ts_tree) <= unzip2 (cla_adders ts ts_rep)
    ts_rep <= cons (scanr fcarry t_rep ts_val)
    t_val <= redr1 (@) ts_val

cla_adders [] ts_rep
 => []
cla_adders (t1:<ts) (t1_rep:<ts_rep)
 => (cla_adder1 t1 t_rep):<(rec ts ts_rep)

cons (a,x) => a:<x
```

A schematic interpretation of an instance of this description (four children per node) in conjunction with the `ListCat` isomorphism of section 3.4.2, is given in figure 5.3.3, where the following instance of `cla_adders` is assumed:

```
cla4 ([(p3,g3),(p2,g2),(p1,g1),(p0,g0)],c)
 => ((p,g),[c3,c2,c1,c0])
    where
    c0 <= c
    c1 <= g0 \/ (p0 & c)
    c2 <= g1 \/ (p1 & g0) \/ (p1 & p0 & c)
    c3 <= g2 \/ (p2 & g1) \/ (p2 & p1 & g0) \/
          (p2 & p1 & p0 & c)
    p <= p3 & p2 & p1 & p0
    g <= g3 \/ (p3 & g2) \/ (p3 & p2 & g1) \/
         (p3 & p2 & p1 & g0)
```

In realizing the carry-lookahead adder one must be careful so that the implementations of `scanr` and `redr1` have the same number of logic levels (assuming that the realization of operators `@` and `$` have the same propagation delays). This will help in the pipelining of the resulting tree structures, a topic which is discussed in the next section.

## 5.4 Retiming Temporal Attribute Grammars

One of the problems with the tree-like circuits derived using the laws of the previous section is that, when they communicate with synchronous systems, their clock period must be long enough in order to allow signals to ripple through the entire tree. It would be preferable if the clock period was independent of the size of the tree and depend only on the time required for a signal to propagate through a single functional element, even if this resulted in increased latency. Systems with this property are called systolic and they have been studied extensively during the last 15 years. Many methodologies have been suggested for deriving systolic systems from "regular" synchronous systems, the canonical mapping and the cut-set systolization procedures among others [46]. A sufficient condition for a synchronous system to be systolic is that every communication path between functional elements is intercepted by at least one register. The essential property that allows us to move registers around is shift-invariance (discussed in section

3.6.2). In this section we investigate the extent to which our algebraic framework can be applied in this kind of situation.

Returning to our bidirectional trees on the right-hand-side of laws 5.3.1 and 5.3.2, we observe that, if we abstract the combination of up and dn into a single functional element corresponding to the operations performed at each `Cat` node, then it is impossible to systolize the resulting system by any shifting of registers around them. In fact, these systems are not even synchronous, i.e. they have cycles of zero weight! Nevertheless, after examining the compound dependency graphs, we know that these systems are well-defined. The major insight, then, that allows us to perform the systolization is to regard the part(s) computing the synthesized attributes independently from the part(s) computing the inherited ones. The following generic theorem, when instantiated with `D` in place of `k`, can be used to pipeline bidirectional tree—shaped architectures. Before that, though, we need to define the `depth` of a tree (assuming the predefined operations `+` and `max`):

```
depth Nilt       => 0
depth (Unit a)   => 0

depth (x Cat y) => 1 + (rec x) max (rec y)
```

**Theorem 5.4.1** For all perfectly balanced trees[6] `t`, when `k` distributes backwards over `@` and `$` (that is, when `k (a @ b) == k a @ k b` and `k (a $ b) == k a $ k b`), then:

```
(map (repeat k (2*d-1)) || repeat k d)
(catAG f up r t e)
==
catAG f1 up1 r dn1 t (repeat k (d-1) e,0)
where
d <= depth t
r ti => (Nilt,e)
f a ti => (Unit ti,a)
((xt,xv),(yt,yv)) up ti => (xt Cat yt, xv @ yv)
((xt,xv),(yt,yv)) dn ti => (ti, ti $ xv)
f1 a (ti,h) => (Unit ti,a)
((xt,xv),(yt,yv)) up1 (ti,a)
 => (xt Cat yt,k (xv @ yv))

((xt,xv),(yt,yv)) dn1 (ti,h)

 => ((k ti,h+1),(k (ti $ (repeat k (2*h) xv)
```

A schematic interpretation of the above theorem in case that `k` is instantiated by a unit delay is given in figure 5.4.1.

In any practical implementation, it is very unlikely that the inherited attribute `h` on the right-hand-side of the above law will be computed using the above equations[7]. What is more likely is that the distance of each node from the root will be pre-computed and "stored" in each particular processing node making them, of course, non shift-invariant.

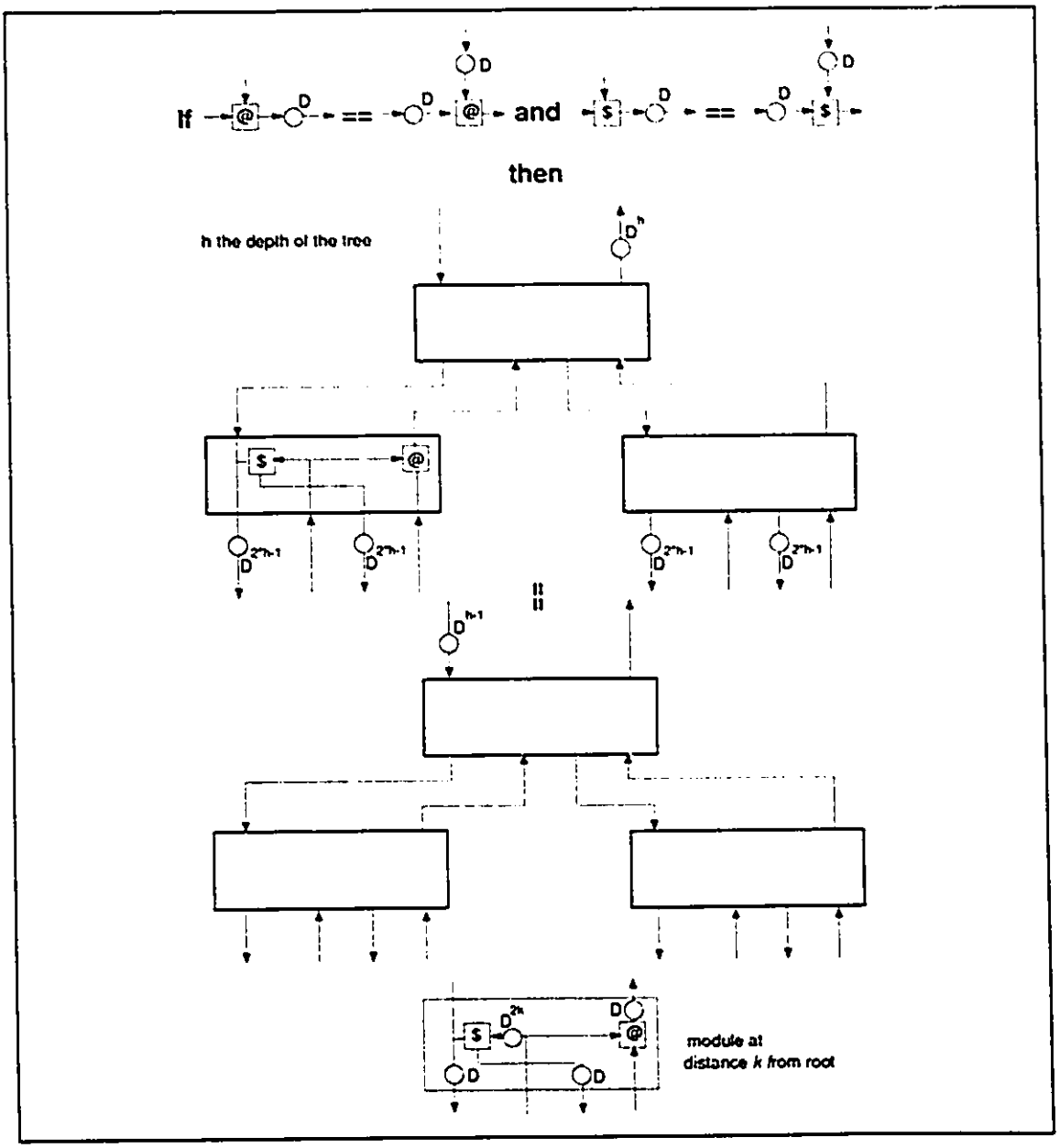
Theorem 5.4.1 can be proved by "subgoal induction" (see section 2.3.2 of [56]) on the depth of the recursion `h` and subsequent (laborious) case analysis. Instead of proving this single theorem though, we are going to provide an even more powerful procedure that can be used to derive a systolic schedule for any arbitrary TAG, provided that this is possible, or report an error in

[6] If the tree is a view of a list obtained using the `list2cat` and `cat2list` coercions of section 3.4.2, then `t` is perfectly balanced if `length (cat2list t)` is a power of 2.

[7] Notice that the distance `h` of a term `a` from the root of a tree `t` can be also computed using synthesized attributes only as `h = depth t - depth a`.

case that it is not. It is inspired by Leiserson and Saxe's systolic conversion theorem [78] which is widely used in the industry, and addresses the main criticism that people have against it, namely that the algorithm that constructs the systolic schedule (single-destination-shortest-paths procedure) does not take into account the regularity that exists in most of the data parallel systems.

Figure 5.4.1. Retiming of generic (perfectly balanced) *Cat* tree TAGs



**TAG retiming procedure:** Given a well defined temporal attribute grammar $T$ with compound dependency graph $G$, construct the *constraint dependency graph G-1* which is obtained by replacing every edge $(a1,a2,w)$ in $G$ by $(a1,a2,w\text{-}1)$. Then:

1. If $G\text{-}1$ has cycles with negative sum of weights, report that no retiming is possible without using specific properties of the operators involved in the attribute evaluation rules of $T$ and STOP, otherwise proceed to step 2.
2. Assign the following *lags* to each attribute occurrence with respect to the root of the grammar:
   a. Every inherited attribute $aI$ at distance $h$ from the root will get a $lag\ (aI) = h\text{-}1$.
   b. Every synthesized attribute $aS$ at distance $h$ from the root will get a $lag\ (aS) = -(h\text{-}1)$.
3. Replace every edge $(a1,a2,w)$ in $G$ by $(a1,a2,lag\ (a1) - lag\ (a2) + w)$ obtaining $G'$. Every weight in $G'$ is now strictly positive.

The TAG $T'$ obtained by retiming using $G'$ has the same behavior as the original TAG $T$ and it is systolic.

**Proof** The TAG retiming procedure is going to be justified by referring to the *Retiming Lemma* and the *Systolic Conversion Theorem* of Leiserson and Saxe [78]. In a TAG, all communication with the outside world (host) takes place through the root. The dependency graph corresponds to a family of *communication graphs*. If the constraint dependency graph of the TAG has no cycles with negative sum of weights, then (by induction) the same is true of the communication graph of any derivation tree constructed using this TAG. Therefore, the condition

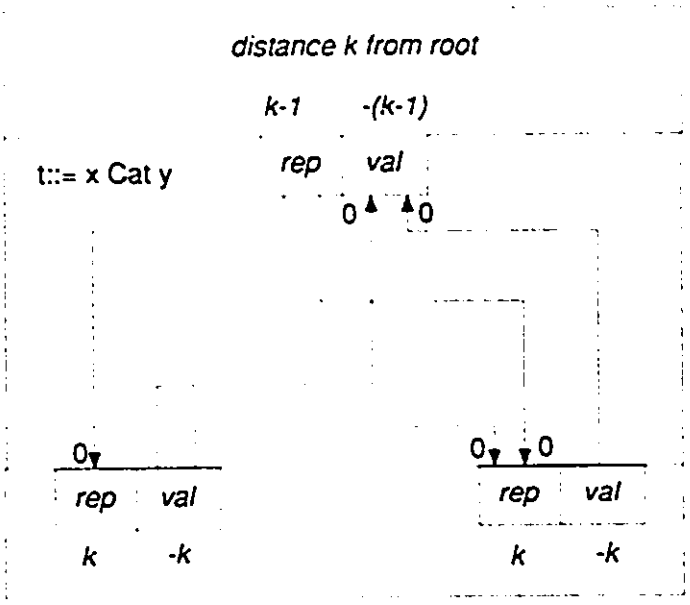
of step 1 (existence of a systolic schedule) is equivalent to the condition of the Systolic Conversion Theorem.

By definition, now, synthesized attributes are directed towards the root of the attributed tree and inherited attributes are directed away from it. If a node at distance $h$ from the root contributes to the output (distinguished synthesized attribute of the root), then it must *lead* the root by $h-1$ clock cycles because this is the *shortest path* between them. Conversely, if a node at distance $h$ wants to make use of data broadcasted from the root (inherited attributes), it must *lag* the root by $h-1$ clock cycles because this is the shortest path between them. There is no other way in which two nodes of a TAG can exchange information. This justifies step 2 in the above procedure. By virtue, then, of the Retiming Lemma, the schedule constructed in step 3 is going to produce a system $T'$ which has the same behavior (modulo the initial $k$ clock cycles, where $k$ is the latency of the system, since our registers $D$ have unknown initial values). The Systolic Conversion Theorem then guarantees that $T'$ is a systolic system.

In our assessment, the greatest advantage of the retiming procedure is that it does not change the connectivity of the original TAG. Therefore, if the original TAG had regular topological properties, these will be carried over into the retimed TAG. This separation of concerns, obtaining a regular layout and then retiming in order to systolize it, is of particular importance in the design of regular arrays.

As an application of the TAG retiming procedure, consider the `scanIt` described in previous sections. It is easy to see by inspecting its constraint dependency graph that no negative sum cycles exist, therefore retiming is possible by assigning a lead of $k-1$ (or, equivalently, a lag of

Figure 5.4.2. Step 2 of the retiming procedure: assigning lags to attribute occurrences




$-k+1$ ) to the synthesized attribute *val* at distance $k$ and a lead of $k-1$ to the inherited attribute *rep* at distance $k$ from the root (see figure 5.4.2 where the dependencies for *tree* have been removed in order to avoid unnecessary detail). The compound dependency graph of the retimed TAG constructed using the above procedure is shown in figure 5.4.3 (only the most interesting case of the `Cat` nodes is shown). All attribute dependencies are now positive, therefore the retiming has been successful in producing a systolic system with equivalent behavior. The retimed TAG itself is obtained by back-annotating the original TAG with the new weights and is shown in figure 5.4.4.

As a final remark, we note that even if the constrained dependency graph have cycles of negative weight, we may be able to retime a *slown-down* version of the original TAG by applying techniques similar to that of [115]. This will not be explored further in this thesis.

## 5.5 Concluding comments

We have found that the description of iterative circuit structures using Temporal Attribute Grammars has a number of advantages:

Figure 5.4.3. Retimed compound dependency graph of the *scanlt* TAG

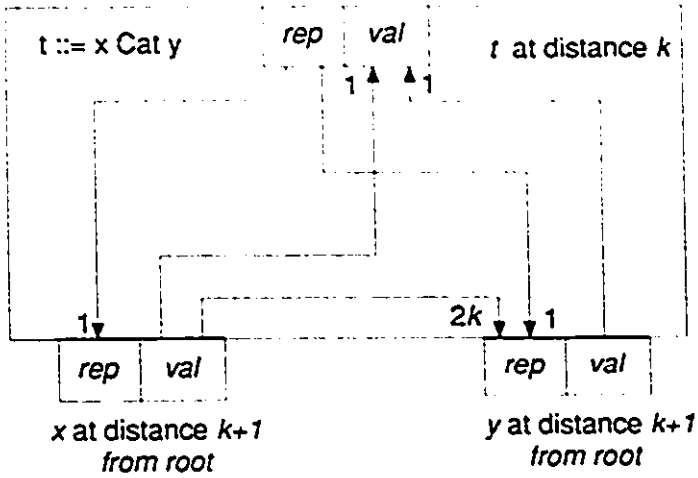
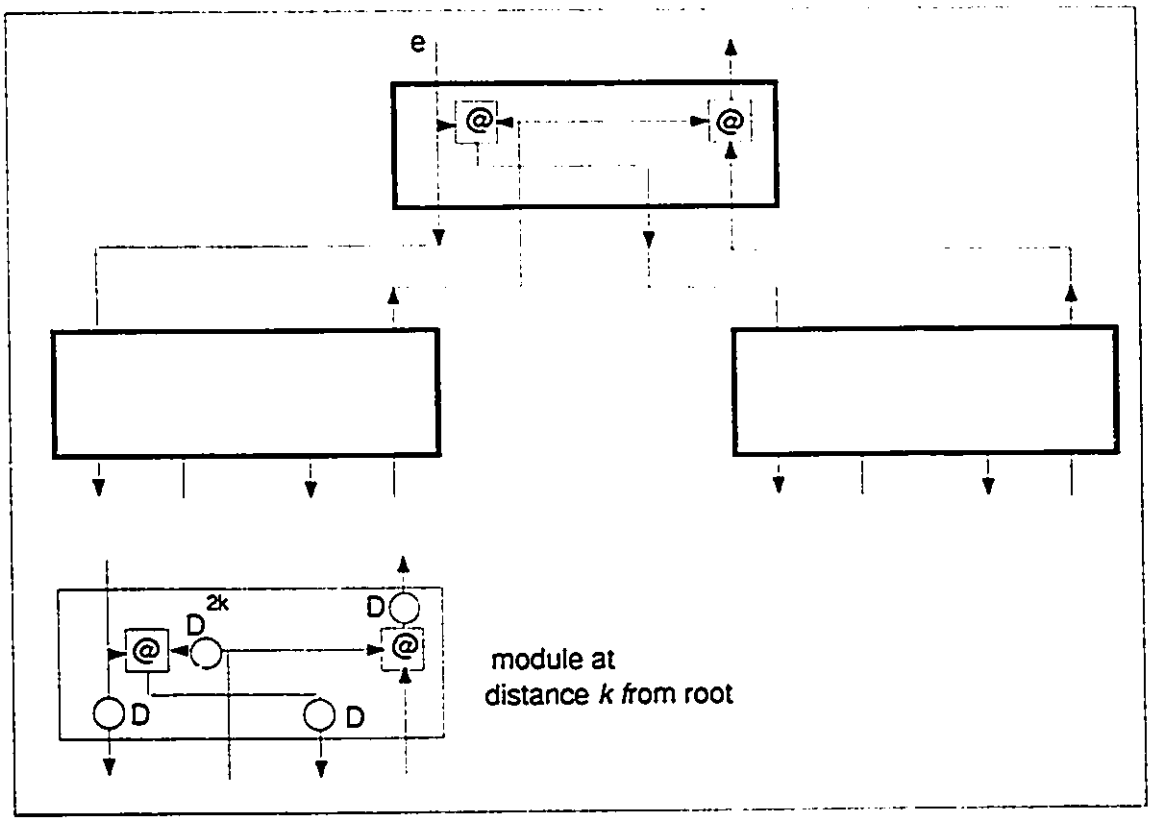



Figure 5.4.4. An instance of the fully pipelined *scanlt*




□ Locality: information is exchanged between modules through well defined attributes. In [79], a two-step procedure is proposed for the construction of systolic systems: first design a *semisystolic* system and then eliminate all instances of global communication (broadcasting of data

and rippling logic) by using *transmittent* data (which corresponds to what are called *copy rules* in attribute grammar terminology) and *collecting* the values computed by the rippling logic using a *breadth-first* spanning tree of the connection graph (undirected and unweighted communication graph). But these are exactly the properties of a well designed attribute grammar!

- The decomposition of an algorithm as an attribute grammar gives us a much better operational "handle" in order to study the implications of choosing a target architecture. In particular, introduction of inherited attributes results in "defunctionalization" of the original algorithm making it, in a sense, more implementable in hardware: the higher-order functionality is replaced by functional dependencies between inherited and synthesized attributes.
- Regularity: the "shape" of the resulting circuit follows the function that is been computed and this, in turn, follows the shape (view) of the input data structure. In our opinion, no other notation exposes better the correspondence between form and function than attribute grammars.
- Parallelism: TAGs are purely data-driven as opposed to other formal derivation techniques based on continuations [57] which are control-oriented and, therefore, geared more towards uniprocessor architectures. It must be noted also that if the TAG is *L-ordered* (or, in general, visit-oriented) [41], then an evaluation sequence can be deduced statically and, therefore, a finite automaton directed by the visit-sequences (visits to subtrees or visits of the context) and a push-down stack for storage of current visits (whose height is at most that of the tree) can be constructed. As a consequence, the same results with continuation-based transformations can be obtained for serialized architectures.
- Our retiming algorithm is more comprehensive than the retiming laws developed by [115]. In particular, it is impossible to pipeline the tree architectures developed in this chapter by using RUBY's retiming laws.

Given the examples discussed in this chapter, it should be wrong to conclude that the TAG paradigm is only applicable to tree-like structures. In fact, left reductions or left accumulations

cannot be defined as homomorphisms on cons lists. They have to be defined as parameterized homomorphisms, i.e. TAGs, for example, see the definition of `rowl` in section 4.2.1. It should be considered as an extra bonus of our notation for inductive variables that it reveals this extra information about the structure of our definitions. It turns out that the *lambda-hoisting* transformation [123] can be fully developed within the framework presented in this chapter. In fact, we believe that Temporal Attribute Grammars are the ideal vehicle for studying recursive filters (IIR) and other algorithms involving circular data structures.

# Chapter 6 SWITCH-LEVEL TRANSFORMATIONS

In chapters 4 and 5 we presented a set of transformation rules based on parameterized *free* type structures. The objective of the work reported in this chapter is to extend this design calculus towards more "concrete" implementation realms requiring the use of *dependent* data types [52]. For this to be realized, we have to define switch-level models for the new computational basis (MOS transistors) as well as some new notions of circuit equality (and inequality), the formalization of which, we claim, is very natural within our framework.

## 6.1 Introduction

A CMOS circuit is a collection of transistors connected to each other by wires. The intuitive explanation of the operation of a CMOS transistor is that of an ideal switch with only two states, *on* or *off*, all other intermediate states are ignored. The ability to drive other gates and detection of short circuits are usually considered as concerns orthogonal to the establishment of correct design functionality. Using this line of reasoning, a number of design techniques have been proposed in the literature, switching trees [109] and C-graph models [24] amongst others. These methodologies are ad-hoc in the sense that they require a global analysis of the circuit topology and, thus, they are not appropriate for hierarchical circuit development and top-down design decomposition. It is our belief that for formal design techniques to be accepted by the VLSI community, we have to extend their realm into technology specific issues, which is today almost synonymous to CMOS. This work is a contribution towards this goal.

Modelling physical devices is an activity which is quite distinct from other circuit design activities, such as digital design. In the case of digital design, there exists a well established mathematical

model, i.e. boolean algebra, that can be used for formal reasoning. In the case of transistors, a widely accepted mathematical basis has not yet been established. The designer has to invent (or choose) an "accurate" model of behavior that is appropriate for the intended application. The *more* detailed the model, the *less* restrictions it imposes on the circuit's operating environment. Proofs that are carried out at the more detailed level also hold for circuits involving less detailed models but the inverse is not always true. The most detailed model (least restricted) is naturally the analog one, e.g. the SPICE model; unfortunately, today's formal synthesis and verification technology does not allow us to manipulate (symbolically) the differential equations used to describe the transistors at this level. One can only hope that a compromise can be found, one that imposes "reasonable" restrictions on the circuit's operating environment but which is mathematically tractable at the same time.

The key idea used in our transistor model is to de-couple the verification of functional behavior from other issues, such as quality or performance. This separation of concerns is not new, it is standard engineering practice: first design a system that "works correctly" and then try to improve its characteristics. What is new is the use of the particular *vertical*[1] constraint abstraction and propagation schemes employed in our models so that design refinement can be performed in a top-down fashion.

As in chapter 3, we will formalize this approach in terms of system semantics. The structural semantics are independent of the choice of the atomic components, therefore we will proceed by presenting the behavioral semantics of CMOS circuits.

## 6.2 Behavioural semantics of CMOS circuits

The atomic components of a MOS circuit are N and P-type transistors, the power supply and the ground; charge capacitances are considered as part of the transistors for the purpose of this

[1] They are vertical (a term first coined by Eveking [45]) in the sense that they cross the gate to switch-level abstraction border.

discussion. In ASDL:

```
atom Pwr :: Wire
atom Gnd :: Wire
atom Ntran :: (Wire,Wire) -> Wire
atom Ptran :: (Wire,Wire) -> Wire
```

The convention that we adopt is that the first position of the `Ntran (Ptran)` corresponds to the gate, the second to the source and the output corresponds to the drain of the transistor.

Combining these atomic components using the facilities of ASDL, unidirectional CMOS networks can be formed. Winskel [136] shows that these kind of networks can never result in a short circuit. Buses with an associated resolution function are used to model situations where there is "contention" of signals. In ASDL:

```
join Join :: (Wire,Wire) -> Wire
```

It turns out that the corresponding definition of the bus resolution function is associative, therefore we can avoid some brackets if we make `Join` an infix operator with less priority than function application.

Finally, since we want to allow some form of syntactic reflection in our structural descriptions, **factored boolean forms**[2] are also admitted as atoms. In ASDL:

```
atom And :: [Wire] -> Wire
atom Or  :: [Wire] -> Wire
atom Not :: Wire -> Wire
```

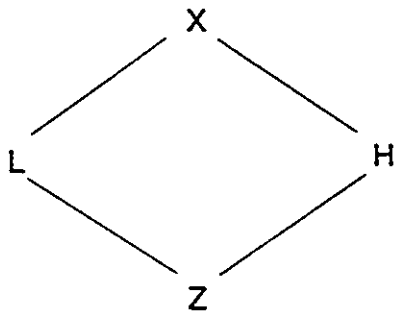
In subsequent sections, we will refer to all of the above atoms and join nodes as MOS terms. One of the key issues in defining a successful semantic function in system semantics is the degree of compositionality of this function, i.e. how easily it can be distributed across subcircuits. In the following, we will introduce the underlying assumptions and simplifications so that a compositional model of CMOS transistor behavior can be introduced.

---

[2] A factored boolean form is, essentially, a generalization of the sum-of-products form allowing arbitrary (but finite) nesting; in general, it doesn't have a canonical form. It has the advantage that it is more compact than sum-of-products and it is very well suited to complex-gate CMOS implementations because the complement of a factored form is also a factored form [131].

### 6.2.1 Instantaneous model

In this model of static behavior, it is assumed that the transistors switch immediately after a new value appears at the gate port. This is similar to the instantaneous model of combinational circuits, therefore local data definitions are not allowed to be recursive for reasons explained in 3.6.1. The domain of interpretation includes the values `L` for low (connected to ground), `H` for high (connected to power), an explicit error value `X` (circuit failure) that denotes connection to both power and ground and a high impedance state `Z` (absence of connection to either power or ground), arranged in the following semi-lattice with a bus resolution function defined on it (the least upper bound):




```
data MOSval = Z | L | H | X

join :: MOSval -> MOSval -> MOSval
join a b = if a<=b then b else if b<=a then a else X
```

The meaning function that deals with the propagation of MOS values is defined inductively on MOS terms. As with all semantic functions of the instantaneous model, it is primed in order to differentiate it from the corresponding functions in the discrete time model. We will make use of an auxiliary environment domain, `Env` (bindings of MOS states to free inputs). Using Haskell as the metalanguage, we have:

```
eval' :: MOSterm -> Env -> MOSval
```

```
eval' Pwr e = H
eval' Gnd e = L
eval' (Ntran (g,s)) e
 = if eval' g e /= L then eval' s e else Z
eval' (Ptran (g,s)) e
 = if eval' g e /= H then eval' s e else Z
eval' (x Join y) e = join (eval' x e) (eval' y e)
eval' (And xs) e
 = if all (==H) (map (\x -> eval' x e) xs) then H else L
eval' (Or xs) e
 = if all (==L) (map (\x -> eval' x e) xs) then L else H
eval' (Not x) e
 = if eval' x e == H then L else H
```

The propagation of values defined in the above model is very similar to the one suggested in [24]. We realize now that the boolean abstraction function of a MOS value is a partial one: some MOS terms evaluate to a value different from either `H` or `L`. The situation is even more complicated because the above model of transistor behavior fails to account for signal degradation through a channel: a N-transistor behaves as an ideal switch for `H` values only, whereas a P-transistor behaves as an ideal switch for `L` values only. In order to account for different signal strengths (that rank the reciprocal of the Thevenin equivalent resistance of a node [101]) a more complicated value system can be used, as is the case with switch-level simulators where sometimes 64 distinct values or more are used. This may be necessary for NMOS circuits but it would be an overkill for CMOS circuits where two additional states are deemed sufficient to express the distinction between driven and undriven wires. We use an additional *projection* for a MOS state, a `Drive` projection. Any two valued domain will suffice but we have nevertheless decided to identify `Drive` with `Bool`:

```
type Drive = Bool
type MOSstate = (MOSval, Drive)
```

The following predicate, called the **drive condition**, depending on both projections of `MOSstate` (unlike `eval'`), is defined as a quality metric of a MOS term in a given operating environment. It is true when the drive projection of the output is true and its value is boolean, i.e. `H` or `L`. The

*weakest precondition* for this to happen is defined as follows[3]:

```
driven' :: MOSterm -> Env -> Drive
driven' Pwr e = True
driven' Gnd e = True
driven' (Ntran (g,s)) e
 = driven' g e && eval' g e == H && eval' s e == L &&
   driven' s e
driven' (Ptran (g,s)) e
 = driven' g e && eval' g e == L && eval' s e == H &&
   driven' s e
driven' (x Join y) e
 = (driven' x e || driven' y e) && valid' (x Join y) e
driven' (And xs) e   = and (map (\x -> driven' x e) xs)
driven' (Or xs) e    = and (map (\x -> driven' x e) xs)
driven' (Not x) e    = driven' x e
```

In words, the above equations state the fact that the power supply lines are driven. For N-type transistors, the drive of the source is propagated to the drain if a driven signal is applied at the gate and the value to be propagated from the source is low, assuming that the transistor is "on" ; similarly for P-type transistors. Notice that drive is never propagated from outputs to inputs. A `Join` node is driven if either of the joined wires is driven and, in case that are both driven, the value of the join has to be "valid" as well, a term that is going to be defined next. When we hide a wire, the drives are propagated together with the MOS values[4]; this is an advantage that the applicative style has over logical formulations [54] where, owing to the fact that the drive of a hidden node is not known, one has to assume the worst case and weaken the specification of the drive condition. Finally, the boolean gates produce strongly driven outputs provided that their inputs are also driven.

The next step is to make sure that a CMOS circuit does not operate in environments that create permanent connections between power and ground (which would cause perpetual flow of current), i.e. to disallow error values from appearing at its outputs. The precondition for this to happen is

---

[3] In Haskell, `not` is the symbol for logical negation (different from the MOSterm `Not`), `/=` denotes inequality, `||` denotes logical or, and `&&` denotes logical and.

[4] No recursive data definitions are allowed in this static behavioral model for obvious reasons.

the electrical **validity condition** (also called "adequacy condition" in [54]) and is derived from the definitions of `eval'` and `driven'` given above:

```
valid' :: MOSterm -> Env -> Bool
valid' Pwr e = True
valid' Gnd e = True
valid' (Ntran (g,s)) e = valid' s e
valid' (Ptran (g,s)) e = valid' s e
valid' (x Join y) e
 = valid' x e && valid' y e &&
   not (ex == H && ey == L) && not (ex == L && ey == H)
   where
   ex = eval' x e
   ey = eval' y e
valid' (And xs) e  = and (map (\x -> driven' x e) xs)
valid' (Or xs) e   = and (map (\x -> driven' x e) xs)
valid' (Not x) e   = driven' x e
```

Note that the validity conditions are propagated in a direction opposite to the propagation of MOS values, that is from outputs to inputs. In words, the equations shown above state the fact that the power supply lines taken in isolation cannot create a short. For a (N or P) transistor to operate without failure, the source must be in an environment that assigns one of the three values `L`, `H` or `Z` (the tristate set) to it; this simplification is made possible by the fact that we don't care whether a non-conducting transistor is valid or not. The joining of two outputs does not create a short-circuit provided that both belong to the tristate set with the additional requirement that if both evaluate to a boolean value, then these values must be equivalent. Finally, boolean gates do not fail provided that their inputs are all driven; this may create "false errors", but it is probably better to err on the side of pessimism in order to account for all possible gate implementation technologies.

As a simple application of the above, consider the archetypal example of a CMOS circuit, the inverter. Its structural description is:

```
inv x => a Join b
         where
         a <= Ptran (x,Pwr)
         b <= Ntran (x,Gnd)
```

First, we calculate the validity condition for the inverter (we use the convention in the justification of a step that the unfolding of a semantic function is annotated with the MOS term on which it is applied):

```
valid' (inv x) e
{unfold inv, introduce abbreviations (1)}
= valid' (a Join b) e'
  where (1)
  e' = e ++ [(a,(ea,da)), (b,(eb,db))]
  ea = eval' a e'
  da = driven' a e'
  eb = eval' b e'
  db = driven' b e'
{valid' Join, (1), introduce abbreviations (1v)}
= va && vb && not (ea==H && eb==L) && not (ea==L && eb==H)
  where (1v)
  va = valid' a e'
  vb = valid' b e'
{case analysis of ea==L BY eval' Ptran, (1), introduce (2)}
= va && vb && not (ea==H && eb==L) && not (ex/=H && eval' Pwr e'==L &&
eb==H)
  where (2)
  ex = eval' x e'
  dx = driven' x e'
{case analysis of ea,eb BY eval' Ptran, eval' Ntran, (1v), (2)}
= not (ex /= H && eval' Pwr e' == H && ex /= L && eval' Gnd e' == L)
```

```
{eval' Pwr, eval' Gnd, simplification, DeMorgan, (2)}
= ex==L || ex==H
{(1), (2), eval' x e' == eval' x e}
= eval' x e == L || eval' x e == H
```

That is, the CMOS inverter does not fail in an environment that assigns H or L to its input. Using the same local definitions as above, we can now derive the drive condition for the CMOS inverter:

```
driven' (inv x) e
{definition of inv, (1) above}
= driven (a Join b) e'
{driven' Join }
= (driven' a e' || driven' b e') && valid' (x Join y)
{derivation of valid' above}
= (driven' a e' || driven' b e') && (eval' x e == L || eval' x e == H)
{driven' Ptran,Ntran, (1),(2) above}
= (dx && ex==L && eval' Pwr e'==H && driven' Pwr e' ||
  dx && ex==H && eval' Gnd e'==L && driven' Gnd e') &&
 (eval' x e == L || eval' x e == H)
= driven' x e
```

In words, the inverter is driven when the input is a strongly driven boolean value. We can see that the drive condition implies the validity condition, i.e. if the inverter is placed in an operating environment that results in strong outputs, then this also guarantees that the CMOS inverter will not fail. It is a simple exercise now to show (using case analysis on `eval'`) that the CMOS inverter satisfies its functional specification as well, i.e. `eval' (inv x) e == eval' (Not x) e` when `x` is boolean.

We have now assembled enough machinery to be able to define what it really means for a CMOS

circuit to be *better* than another one. It makes sense to think of the behavior of a CMOS circuit as a (potentially partial) function `MOSstate->MOSstate`. The drive condition, then, corresponds to co-domain restrictions such that the output is strongly driven. It is a fact of CMOS design that a driven wire is more useful (has better performance) than an undriven one. Therefore, the better circuit will have the *stronger* drive condition. The satisfaction of the drive condition *implies* the satisfaction of the validity condition as well (as can be easily verified by inspecting the equations defining these predicates). On the other hand, the validity condition corresponds to domain restrictions such that the corresponding behavior is defined. Between two circuits that have the same driven states, the better is the one that has the *weaker* validity condition, because this imposes less constraints upon its operating environment. Naturally, under the most "relaxed" conditions (strongest validity condition and weakest drive condition), both circuits must exhibit the same behavior. The above discussion is made more precise with the following definition, where the usual ordering of boolean values (`False <= True`) is assumed:

**Definition** *(Ord MOSterm)* Given two MOS terms `C1` and `C2`, we say that `C1 <= C2` if and only if *(iff)*, for all environments `e` the following propositions are true:

I. `valid' C1 e >= valid' C2 e`

II. `driven' C1 e <= driven' C2 e`

III. `eval' C1 e == eval' C2 e when driven' C1 e`

In a set theoretic interpretation of the above propositions, the better circuit (C2) will have a larger space of valid states and its set of driven states will be a subset of the driven space of the worst one (`C1`)[5]. When the above conditions are simultaneously true, we can substitute C2 for `C1` in an enclosing circuit without imposing any additional constraints on the circuit's operating

[5] Conditions I and II correspond exactly with the type-theoretic definition of the subtype relation in [29], p. 508–511, i.e. the type of C2 is a subtype of C1's type.

environment, in other words, (<=) is a monotonic operator on MOS terms. The CMOS inverter defined above has this property, i.e. `Not <= inv`.

*Ord MOSterm* is useful when verifying circuits whose specifications contain don't care values. When a circuit is completely specified, then simple equivalences suffice, i.e. `C1 == C2` *iff* `C1 >= C2 && C1 <= C2`. This motivates the following definition:

> **Definition** *(Eq MOSterm)* Given two MOS terms `C1` and `C2`, we say that `C1 == C2` *iff*, for all environments e, the following propositions are true:
>
> I. `valid' C1 e == valid' C1 e`
>
> II. `driven' C1 e == driven' C2 e`
>
> III. `eval' C1 e == eval' C2 e when driven' C1 e`

The above definition provides a decision procedure that can be used for the verification of CMOS transistor networks and, also, forms the basis for the behavior preserving transformations rules introduced in section 6.3.

### 6.2.2 Discrete time model

As we have seen in 3.6.2, there are at least two equivalent formulations of a system's dynamic behavior, the stream and the discrete time model. The latter is the preferred one for the presentation in this section. In the discrete time behavioral model, the domain of interpretation is `Nat -> MOSstate` where `Nat` denotes positive integers. Every wire has a value that now depends not only on the environment but on the specific time instance as well. The assumption is that enough time is allowed for the propagation of signals to complete during a single *cycle* of operation; also, due to capacitive effects, wires connected to gates of transistors retain as charge the value that they had at the end of the previous cycle. It is the responsibility of the operating environment to ensure that these conditions are met. This also avoids some of the problems of

*charge sharing* which is outside the scope of our theory. The propagation of MOS values is a straightforward "lifting" of the instantaneous one:

```
eval :: MOSterm -> Env -> Nat -> MOSval
eval Pwr e t = H
eval Gnd e t = L
eval (Ntran (g,s)) e t
 = if eval g e t /= L then (eval s e t) else Z
eval (Ptran (g,s)) e t
 = if eval g e t /= H then (eval s e t) else Z
eval (a Join b) e t = join (eval a e t) (eval b e t)
eval (And xs) e t
= if all (==H) (map (\x -> eval x e t) xs) then H else L
eval (Or xs) e t
 = if any (==L) (map (\x -> eval x e t) xs) then L else H
eval (Not x) e t
 = if eval x e t == H then L else H
```

The recursion restriction in local definitions can be lifted now, provided that the synchronous design rule (no delay-free loops) is satisfied. For the correct application of this rule in CMOS circuits it is assumed that each transistor has a unit-delay associated with its gate but boolean gates are still considered as ideal. This means that, for example, we can have self-dependent loops between the drain and the gate of the same transistor but not between drain and source.

The definition of the drive condition is a little more complicated. The conditions for drive propagation for both types of transistors are now weaker than in the instantaneous behavior case: the gate retains as charge its drive for at least one cycle of synchronous operation (assuming a single-phase clocking scheme). Wires are assumed to be undriven initially (at time 0):

```
driven :: MOSterm -> Env -> Nat -> Drive
driven Pwr e t = True
driven Gnd e t = True
driven (Ntran (g,s)) e t
 = (driven g e t && egt == H || driven g e (t-1) &&
   egt == eval g e (t-1)) && eval s e t == L && driven s e t
   where
   egt = eval g e t
driven (Ptran (g,s)) e t
 = (driven g e t && egt == L || driven g e (t-1) &&
   egt == eval g e (t-1)) && eval s e t == H && driven s e t
   where
   egt = eval g e t
```

```
driven (x Join y) e t
 = (driven x e t || driven y e t) && valid (x Join y) e t
driven (And xs) e t  = and (map (\x -> driven x e t) xs)
driven (Or xs) e t   = and (map (\x -> driven x e t) xs)
driven (Not x) e t   = driven x e t
```

Extensions for multi-phase clocking schemes can be formulated as slight variations of this model. Finally, the definition of the validity condition is a straightforward "lifting" of the instantaneous case:

```
valid :: MOSterm -> Env -> Nat -> Bool
valid Pwr e t = True
valid Gnd e t = True
valid (Ntran (g,s)) e t = valid s e t
valid (Ptran (g,s)) e t = valid s e t
valid (a Join b) e t
 = valid a e t && valid b e t &&
   not (ea == H && eb == L) && not (ea == L && eb == H)
   where
   ea = eval a e t
   eb = eval b e t
valid (And xs) e  t = and (map (\x -> driven x e t) xs)
valid (Or xs) e   t = and (map (\x -> driven x e t) xs)

valid (Not x) e   t = driven x e t
```

The definitions for (in)equality of MOS terms given in the previous section can be readily extended to the discrete time model. In the rest of this chapter, we will restrict our attention to combinational circuits only for which the instantaneous behavioral model is adequate.

## 6.3 Transformation rules for combinational static CMOS design

Using the conditions of section 6.2 it is possible to *analyze* the behavior of complex combinational static CMOS circuits assembled with the assumptions underlying our model (unidirectional transistors, use of `Join` nodes to connect more than two outputs together, no feedback). They can be used, also, for *synthesizing* verifiably correct CMOS circuits by requiring that the *introduction* of new transistors or `Join` nodes satisfies the drive condition and the value specification, assuming that their validity condition is also satisfied. Since the conditions of section 6.2 are compositional, the same technique can be used to justify the introduction of complex CMOS

blocks instead of single transistors. In practice, when combining a number of CMOS subsystems, it is conceivable that some of the validity conditions of interior nodes will be satisfied when the drive conditions of the previous stage are met. As in chapter 4, it is useful to identify "structural constraints", that is, to impose restrictions on the way CMOS systems are composed so that the drive and validity requirements of the interior nodes "cancel" each other. One such restricted design style is the **separated cell** style [24], where series-parallel transistor networks are allowed, but nothing else. In the following we prove that this style is correct with respect to the instantaneous model and proceed to identify laws that allow us to further simplify these networks while *retaining* the separated cell style. These laws were, in part, inspired by the work of Basin et.al. [7] where a logic framework is used instead.

Before we proceed with the laws, we define series-parallel networks using the `redr` combinator from chapter 4:

```
n_series (as,x) => redr Ntran x as
p_series (as,x) => redr Ptran x as
para f [a] x     => f (a,x)
para f (a:<as) x => (f (a,x)) Join (rec as x)
n_para (as,x) => para Ntran (as,x)
p_para (as,x) => para Ptran (as,x)
```

All CMOS systems examined in this chapter can be formed using these combinators.

The first two laws allow the introduction of transistors in place of `And`, `Or` gates:

**Proposition** *(ntranI)* Assuming that all inputs `as` are driven, the following equivalences are tautologies in the instantaneous behavioral model:

```
Ntran (And as,x) == n_series (as,x) (ntranIseries)

Ntran (Or as,x)  == n_para (as,x) (ntranIpara)
```

where `length as≥2`

For a detailed proof of this proposition see appendix D. Notice, though, that there is a "pitfall"

in our proof: n can be arbitrarily large and, this suggests that any number of transistors can be connected in series, which is known to cause problems due to excessive delays, charge redistribution and body effect. In reality, this is not a problem of our proof, it is rather a deficiency of our transistor models. This is a reminder of the fact that formal verification methods are as accurate as their underlying models are. The situation can be remedied in our case by imposing the extra design rule that "no more than (say) 6 transistors can be connected in series".

Suppose that the only available CMOS building blocks are the `n_para` and `n_series` combinators. The resulting circuits are called "N-blocks". Owing to proposition *ntranl*, we can always substitute a N-block with a circuit of the following form: `Ntran (fN as,x)`, where `fN` is a factored formula with free variables (inputs) `as`, which are assumed driven, and `x` is the signal to be propagated. We shall call `fN` the **equivalent boolean formula** of a N-block.

The dual of *ntranl* for P-transistors is:

**Proposition** *(ptranl)* Assuming that all `as` are driven, the following equivalences are tautologies in the instantaneous behavioral model:

```
Ptran (Or as,x)  == p_series (as,x) (ptranlseries)
Ptran (And as,x) == p_para (as,x) (ptranlpara)
```

where `length as≥2`

The proof of this proposition is similar to the proof of *ntranl*.

Although *ntranl*, *ptranl* are equivalences, in practice we are going to use them as left-to-right rewrite rules because this is the direction that reduces boolean specifications into transistor networks. If we assume that each `as` is a complex boolean function in factored form, then we can see that each successive application of the above rules reduces the complexity of the formulae that are driving the gates of the transistors without imposing any additional constraints on the operating environment. As a consequence, repeated application of these rules will eventually

terminate with only literals driving the gates of the transistors. These will typically be primary inputs and, therefore, the drive assumption will be discharged. If we want the outputs of our CMOS circuits to be driven, then, after examining the above proofs, it follows that (assuming that the environment asserts the equivalent boolean condition) for the *ntranl* rule, x must be connected to the ground through a "good" path and that for the *ptranl* rule, x must be connected to the power through a "good" path as well. This is the justification for the following rule:

**Proposition** *(net_create)* When x is always driven, then: `f x ==`

```
(Ptran (Not (f x),Pwr)) Join (Ntran (Not (f x),Gnd))
```

where `f` is a completely[6] specified boolean formula.

The proof of this proposition is given in appendix F.

Using first *net_create* and then *ntranl* and *ptranl*, we make sure that zeros are propagated through ideal N-switches and ones through ideal P-switches. Static CMOS design makes also use of fully complementary transmission gates (which provides good paths for both high and low values) and this usually results in an optimized circuit with respect to transistor count. The introduction of pass transistor logic in our circuits is justified with the following proposition:

**Proposition** *(ntranR)* Assuming that `fN` is the equivalent boolean formula of a N-block with inputs `as`, then

```
(Ntran (fN as,Ntran (a,Gnd))) Join x ==
(Ntran (fN as,Not a)) Join x
```

when

```
not (eval' (fN as) e == H && eval' a e == L &&
     eval' x e == L)
```

The proof of this proposition is presented in appendix E.

---

[6] A formula is completely specified if it evaluates to either H or L.

The condition that `not (eval' (fN as) e == H && eval' a e == L && eval' x e == L)` is an example of a **behavioral constraint.** Behavioral constraints are restrictions imposed on the operating environment of a circuit under which the particular abstraction of its behavior is valid. They are complementary to structural constraints and sometimes both kinds are needed in order to define a particular design style. As an example, the Mead & Conway style is characterized by the use of both N-blocks and transmission gates and its formal analysis would therefore require both kinds of constraints.

In the above proposition, the derivation of the equivalent boolean formula requires analysis of the corresponding block. Sometimes it makes sense to restrict our attention to the most common application of *ntranR*, that is when the N-block is simply a series of N-transistors:

**Corollary** *(ntranRseries)* Assuming that all inputs are driven, then

```
n_series ([a0,a1,...an],Gnd) Join x ==
n_series ([a1,...an],Not a0) Join x
```

when

```
not (eval' a1 e == H && ... && eval' an e == H &&
     eval' a0 e == L && eval' x e == L)
```

where n>0

The proof is a straightforward application of *ntranR* and *ntranI*.

We now turn our attention to the P-part of a CMOS circuit. There is, naturally, a dual law for replacing transistors connected to the power with a direct connection to a primary input:

**Proposition** *(ptranR)* Assuming that `fP` is the equivalent boolean formula of a P-block with inputs `as` and `x`, then

```
(Ptran (fP as,Ptran (a,Pwr))) Join x ==
(Ptran (fP as,Not a)) Join x
```

when

```
not (eval' (fP as) e == L && eval' a e == H &&
     eval' x e == H)
```

The proof is similar to that of *ntranR*. An application of this proposition is the following corollary:

**Corollary** *(ptranRseries)* Assuming that all inputs are driven, then

```
p_series ([a0,a1,...an],Pwr) Join x ==
p_series ([a1,...an],Not a0) Join x
```

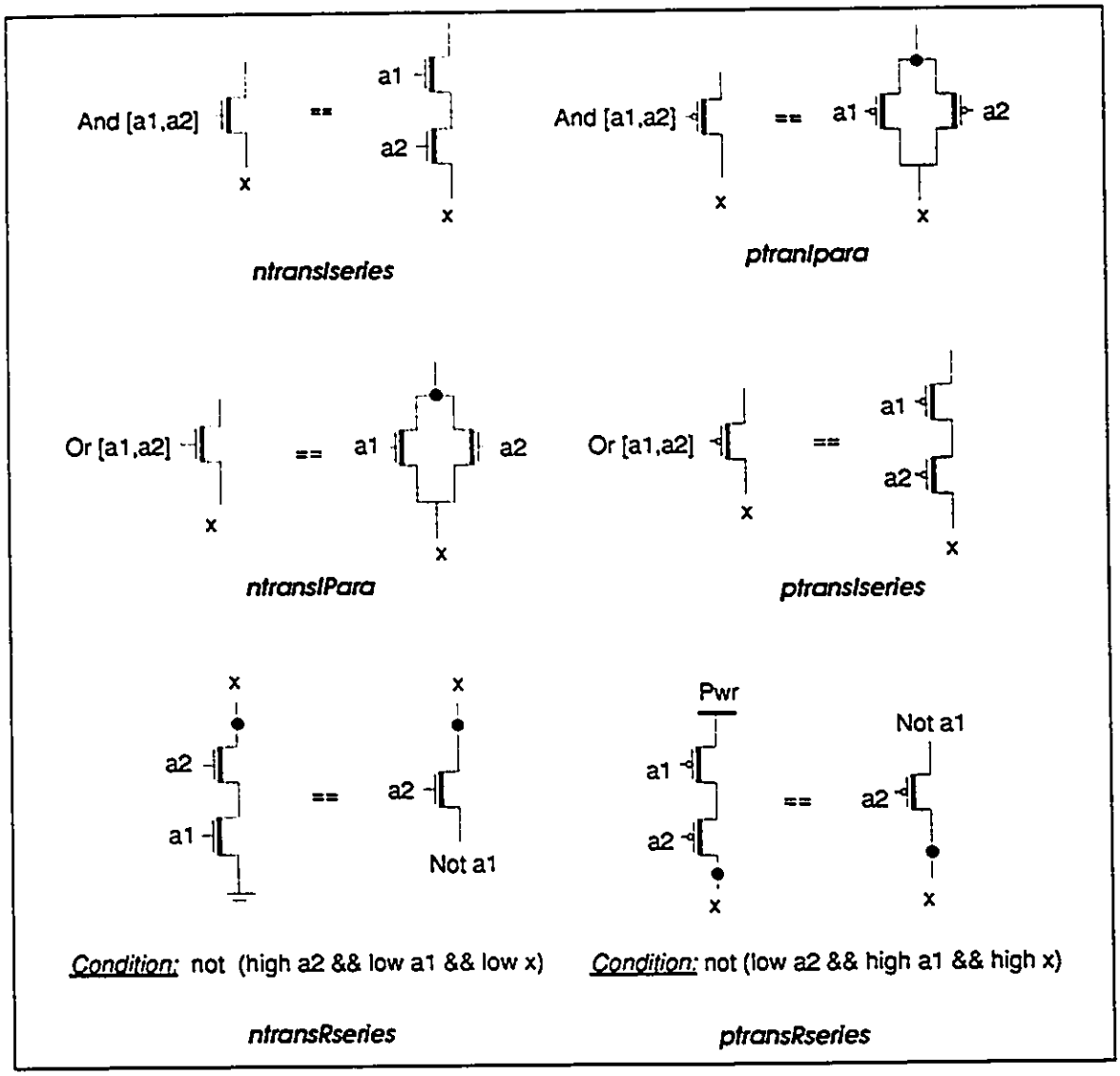
when

```
not (eval' a1 e == L && ... && eval' an e == L &&
     eval' a0 e == H && eval' x e == H)
```

where n>0

An important feature of our transformations laws so far is that they retain the separated cell structure. As a consequence, they can be applied in any convenient sequence. Figure 6.3.1 is a schematic interpretation of the laws restricted to the base case (two inputs only).

Figure 6.3.1. CMOS transformation laws[7]




It should be noted that checking the behavioral constraints for the application of *ntranR* and *ptranR* would require machine assistance even for circuits of moderate complexity. In practice, it is desirable to have transformation laws which are purely *syntactic*, i.e. do not depend on applicability conditions, even at the cost of reduced generality. The following corollary is based

---

7 Solid circles indicate joining of wires. Also, we define `high a = (eval' a e == H)` and similarly for `low`.

on the intuitive idea that multiplexers have a boolean behavior that is *sufficient* to guarantee the satisfaction of the behavioral constraints required by *ntranR* and *ptranR*.

**Corollary** *(mux2I)* Assuming an environment where a, b, c are driven, then

```
mux2 (c,a,b) == (tg (c,a)) Join (tg (Not c,b))
```

where the CMOS transmission gate is defined as:

```
tg (c,x) => (Ntran (c,x)) Join (Ptran (Not c,x))
```

*Proof:* The specification of the multiplexer is given as

```
mux2 (c,a,b) => Or [And [c,a],And [Not c,b]]
```

By applying *net_create* and then repeatedly *ntranI*, *ptranI* we arrive at four transistor chains, each comprised of two transistors, joined together. It is a property of these chains that only one of them can be "on" at any specific time, therefore the conditions of *ntranR* and *ptranR* are trivially satisfied. Note that transistors are eliminated in pairs, therefore resulting in pass-logic networks. Q.E.D.

The proposition introduced above can be generalized to any number of mutually exclusive (but not necessarily exhaustive) cases and, consequently, all the theory that has been developed using data selectors as *universal modules* of boolean algebra can be very naturally carried over to CMOS design. This lemma is important for another reason as well: it provides us with criteria for the optimization of boolean formulae and, therefore, establishes a useful link between logic optimization and CMOS design optimization. This is a point taken seriously by practicing engineers, who try to express their logic specifications in terms of gates with well known (efficient!) static CMOS implementation.

We now state a final law concerning the distribution of a condition in conjunctive form across transmission gates:

**Proposition** *(tgD)* Assuming that all inputs are driven, then for any permutations of a1,...,an, the following formula is a tautology in the instantaneous behavioral model:

```
(n_series ([a1,...,an],x)) Join
(p_series ([Not a1,...,Not an],x) ==
redr tg x [a1,...,an]
```

The proof of this law is a straightforward application of the *muxl* corollary.

### 6.3.1 Examples

The purpose of the examples presented in this section is to motivate the derivation of a deterministic algorithm for synthesis of CMOS networks from boolean specifications. We do not have any claims of novelty here, on the contrary, the circuits derived are standard textbook case studies. The only claim that we have is that our rules capture in a natural way the engineering practice of CMOS design.

We start our presentation with the derivation of the equivalence gate. The derivation style is the same as the one used in previous sections. We assume standard rules of propositional logic:

```
xnor2 [a,b]
{BY net_create}
= xP Join xN
  where
  xP <= Ptran (Not (xnor2 [a,b]),Pwr)
  xN <= Ntran (Not (xnor2 [a,b]),Gnd)
  xnor2 [a,b] => Or [And [a,b], And [Not a,Not b]]
{De Morgan in P-part, De Morgan followed by factorization in N-part}
= xP Join xN
  where
```

```
  xP <= Ptran (And [Or [Not a,Not b],Or [a,b]],Pwr)
  xN <= Ntran (Or [And [a,b], And [Not a,Not b]])
{commutativity of And,Or, BY repeated ntranI, BY repeated ptranI}
= xP Join xN
  where
  xP <= (Ptran (Not a,Ptran (Not b,Pwr))) Join (Ptran (b,Ptran
(a,Pwr)))
  xN <= (Ntran (Not a,Ntran (b,Gnd))) Join (Ntran (Not b,Ntran
(a,Gnd)))
{BY ntranRseries (twice), BY ptranRseries (twice), ...}
= (Ptran (Not a,b)) Join (Ptran (b,Not a)) Join
   (Ntran (b,a)) Join (Ntran (a,b))
```

Note that the "appropriate" use of and-commutativity in the N-block and of or-commutativity in the P-block has saved us from the need to provide inverted inputs for both a and b. An implementation of the exclusive-or function can be easily obtained from xnor2 by substituting a and b with their negations.

We now turn our attention to full-adders and more specifically to the sum part of them:

```
sum (a,b,c)
{definition of sum}
= Or [And [a,b,c],And [a,Not b,Not c],And [Not a,Not b,c],
      And [Not a,b,Not c]]
{definition of xor2, replace common factors with fanout}
= Or [And [c,Not x],And [Not c,x]] where x <= xor2 [a,b]
{definition of mux2}
= mux2 (Not x,c,Not c) where x <= xor2 [a,b]
```

```
{multiplexer law (xor2 [a,b] is total)}
= Not (mux2 (x,c,Not c)) where x <= xor2 [a,b]
{implement Not as an inverter, BY mux2I}
= inv y
  where
  x <= xor2 [a,b]
  y <= (Ptran (Not x,Not c)) Join (Ntran (x,Not c)) Join
       (Ptran (x,c)) Join (Ntran (Not x,c))
```

By substituting with the implementations of `xor2` and `inv` and working along similar lines on the carry part, we obtain the well-known transmission gate adder [66]. A notable feature of circuits derived using the *mux2I* rule is the extensive use of fanout from intermediate nodes.

The derivation of a four input multiplexer is demonstrated next:

```
mux4 (s1,s2,a,b,c,d)
{BY net_create, introduce abbreviations (2)}
= (Ptran (xP,Pwr)) Join (Ntran (xN,Gnd))
  where (2)
  xP <= Not (mux4 (s1,s2,a,b,c,d))
  xN <= Not (mux4 (s1,s2,a,b,c,d))
  mux4 (s1,s2,a,b,c,d) => Or [And [Not s1,Not s2,a],
                              And [Not s1,s2,b],
                              And [s1, Not s2,c],
                              And [s1,s2,d]]
{De Morgan in xP, De Morgan followed by And distributation in xN, (2)}
= (Ptran (And [Or [s1,s2,Not a], Or [s1,Not s2,Not s2],
  Or [Not s1,s2,Not c], Or [Not s1,Not s2,Not d]], Pwr)) Join
```

```
   (Ntran (Or [And [Not s1,Not s2,Not a], And [Not s1,s2,Not b],
   And [s1,Not s2,Not c], And [s1,s2,Not d]], Gnd))
{BY repeated ntranI,ptranI, BY ntranRseries,ptranRseries, BY tgD}
= (tg (s1,tg (s2,a))) Join
   (tg (Not s1,tg (s2,b))) Join
   (tg (s1,tg (Not s2,c))) Join
   (tg (Not s1,tg (Not s2,d)))
```

Notice that the application of *tgD* leaves the transistor count unchanged. Not only are our derivations correct by construction (within the limits of our models) but the complete derivation sequence together with the intermediate results (the record of the design process) are available for latter inspection and for documentation purposes. Finally, notice that the rules for multiplexer introduction can be derived from the other transformation rules as special cases and, therefore, will not be considered in the following algorithm.

## 6.4 A deterministic algorithm for synthesizing combinational static CMOS networks

By studying the derivations of the circuits presented in the previous section, a pattern for the use of transformation rules emerges: first, we specify the circuit in boolean factored form, then we use De Morgan and other Boolean algebra identities to calculate the complement of the boolean specification, then we use rule *net_create* to synthesize a "simple" version of the circuit in complex-gate form, then we optimize independently the N and P-parts of the network by repeatedly applying the rules *ntranI* and *ptranI*. Finally, we optimize the derived circuit by applying the rules *ntranR* and *ptranR*, when the applicability conditions are satisfied. The following algorithm makes this process more precise for single-output circuits:

Algorithm *(static_CMOS_complete)*

**Input:** A *maximally factored* [64] boolean formula `Not fP`, that completely specifies the complement of the circuit to be synthesized, i.e. a formula containing only `And`, `Or`, and `Not` MOS terms. The free variables of this formula are assumed to be the primary inputs of the circuit and are taken to be strongly driven.

**Output:** A static CMOS circuit, i.e. a system containing only `Ntran`, `Ptran`, `Pwr`, `Gnd` and `Not` MOS terms. The last one (`Not`) is allowed in combination with primary inputs only.

Steps:

1. Apply (recursively) De Morgan to `Not fP` deriving `fN`[8]. Although `Not fP` and `fN` are denoted by the same extensional behavior, they are not necessarily defined by the same MOS term expression.
2. Apply rule *net_create* using `fP` and `fN` for the P and N-parts (respectively) deriving `f`; the application of this rule is valid since `fP` is a completely specified boolean formula. No new variables are introduced during this step.
3. Repeatedly apply *ntranI* and *ptranI* to `f` deriving `f'`. This may result in the introduction of new local variables whose names must not cause clashes with previously defined ones. Float all nested local definitions to the top level. The application of these rules is valid since the free variables of `f` are (at most) the free variables of `fP` (primary inputs) which have been assumed strongly driven.
4. In `f'`, examine each path to the ground (power) checking the conditions for the application of rules *ntranR* (*ptranR*). Eliminate transistors for which the applicability conditions are true, otherwise leave the path unchanged.

END

[8] The inverse of a factored form is obtained by exchanging And with Or and modifying the polarities of literals from Not x to x and vice-versa (principle of duality).

*Remarks:*

1. The commutativity of `And`, `Or` operators is not taken into account. This can result in non-optimum solutions with respect to the number of negated primary inputs required (as we have already seen in the `xnor2` example).

2. The efficiency of the algorithm can be substantially improved by partially evaluating the relevant MOS term expressions during step 4 using the constraints implied by the applicability conditions.

3. An even more radical improvement would be to make rules *net_create*, *ntranI* and *ptranI* conditional and eliminate step 4 altogether. This would require, though, a tight integration of steps 2 and 3 which would hinder future extension of the algorithm with additional rules.

4. In comparison with other approaches to CMOS synthesis, this algorithm works synergistically with (multi-level) logic optimization. For example, in case of a complex gate with a large number of inputs and outputs, a realistic implementation would seek to decompose the design task to smaller ones at the cost of introducing additional propagation delays. Since our approach treats transistors as unidirectional devices, any algorithm for algebraic (weak) division [22] can be used to decompose the specification to single output subcircuits that can be synthesized with our algorithm. The definitions of the drive and validity conditions for locally defined signals would then guarantee that the resulting circuit will satisfy its drive and validity conditions under the assumption that the intermediate wires are driven. In contrast, in [24], additional rules for cascading "suitably" interconnected MOS cells had to be introduced in the theory.

As an example application of the last remark, consider the synthesis of a full adder circuit. Since we require that the input be in factored form, we will synthesize a circuit that produces the complements of the carry and the sum from which the non-complemented outputs can be easily derived[9]. We have:

---

[9] This is an instance of the phase assignment problem which has been proved NP-complete in [131].

```
notFA = [not_cout,not_sumout]
{definitions}
= [Or [And [a,b],And [b,cin],And [cin,a]],
   Or [And [a,b,cin],And [a,Not b,Not cin],
       And [Not a,Not b,cin],And [Not a,b,Not c]]
{extract common factors}
= [not_cout,Or [And [not_cout,Or [Or [a,b],cin]],And [cin,And
[a,b]]]]
  where not_cout = Or [And [cin,Or [a,b]],And [a,b]]
```

The synthesis of the two subcircuits results in a CMOS network identical to the one in [134, p. 312].

One disadvantage of the above algorithm is that it fails to take into account the opportunities for optimization offered by the asymmetries of the N and P-part of a CMOS circuit. This fact as well as the quest for an optimization methodology for incompletely specified boolean circuits lead us to the modifications discussed next.

## 6.5 Extensions of the algorithm for dealing with incompletely specified circuits

Incompletely specified boolean functions play a significant role in the theory of boolean logic simplification. Input and/or output don't care values are commonly used in order to further simplify circuits that are guaranteed to operate in a restricted environment. Following [132] we will call these constraints **designer's constraints** in order to differentiate them from the validity and drive conditions. The following algorithm is a modification of the algorithm presented in the previous section which takes advantage of the more "relaxed" conditions afforded by incomplete specifications.

Algorithm *(static_CMOS_incomplete)*

**Input:** Two factored boolean formulae, one, `fP`, defining the conditions that make the output high (the ON set) and the other, `fN`, defining the conditions that make the output low (the OFF set). In addition, a set of designer's constraints, `dc`, on the environment (the complement of the "don't care" set) is given.

**Output:** The same as in algorithm *static_CMOS_complete*.

Steps:

1. Check if `dc` implies that *exactly one* of `fP, fN` is set high. If this condition is satisfied, then proceed to the next step, otherwise report failure and STOP.
2. Apply De Morgan rule on `Not fP` deriving `fP'`.
3. a. Repeatedly apply *ntranI* to `(Ntran (fN x, Gnd)` deriving `N_block`.
   b. Repeatedly apply *ptranI* to `Ptran (fP' x, Pwr)` deriving `P_block`.
4. Synthesize a network `P_block Join N_block` deriving `f`. This implements a circuit which is (at least) as good as its specification (proof follows).[10]
5. Repeatedly apply rules *ntranR* and *ptranR* checking that the relevant applicability conditions are compatible with the designer's constraints `dc`.

END

*Proof:* From the theory of section 6.2, we know that if a circuit satisfies the drive condition then it satisfies the validity condition as well. Hence, according to definition *Ord MOSterm*, in order to show that the circuit implemented during step 4 is as good as the original specification, it suffices to show that the drive condition of the implementation implies `dc`, that is, for all environments `e`:

```
driven' ((Ptran ((Not fP) x, Pwr)) Join (Ntran (fN x,Gnd))) e >= dc e
{...}
```

[10] As an aside, it is easy to verify that by interchanging `fP'` and `fN` we obtain `Not f`.

```
= not (eval' (fP x) e == H && eval' (fN x) e == H) &&
       (eval' (fP x) e == H || eval' (fN x) e == H) >= dc e
{definition of exclusive-or ...}
= (eval' (fP x) e == H) xor (eval' (fN x) e == H) >= dc e
```

But this is exactly the condition checked during the first step. Verifying that `f` has the same functional behavior as the specification `fP` when `dc` is true is trivially done by case analysis. Q.E.D.

As an application of this algorithm, let us examine a four input *barrel shifter* with input data bus `in0,in1,in2,in3`, shift lines `s0,s1,s2,s3` and output bus `out0,out1,out2,out3`, whose specification is given in the following table, with underscore characters indicating don't care values:

Table 6.5.1. The output specification of a four bit barrel shifter

| s0 | s1 | s2 | s3 | out0 | out | out2 | out3 |
|---|---|---|---|---|---|---|---|
| H | _ | _ | _ | in0 | in1 | in2 | in3 |
| _ | H | _ | _ | in3 | in0 | in1 | in2 |
| _ | _ | H | _ | in2 | in3 | in0 | in1 |
| _ | _ | _ | H | in1 | in2 | in3 | in0 |

From this table, the ON and OFF set specifications for the first output are easily derived:

```
out0P = Or [And [s0,in0],And [s1,in3],And [s2,in2],
            And [s3,in1]]
out0N = Or [And [s0,Not in0],And [s1,Not in3],And [s2,in2],
            And [s3,Not in1]]
```

Similarly for the other outputs. We will assume an environment where only one of the selector lines is set high at any particular time, i.e.

```
dc = Or [And [s0,Not s1,Not s2,Not s3],
         And [Not s0,sel1,Not s2,Not s3],
         And [Not s0,Not s1.s2,Not s3],
         And [Not s0,Not s1,Not s2,s3]]
```

Now, it is a simple exercise to verify that in all[11] environments e,

`(eval' (out0P x) e==H) xor (eval' (out0N x) e==H)>=dc e.` Therefore, steps 2–5 of the above algorithm can be applied (by repeating the same procedure for each output) deriving the circuit in figure 6.5.1.

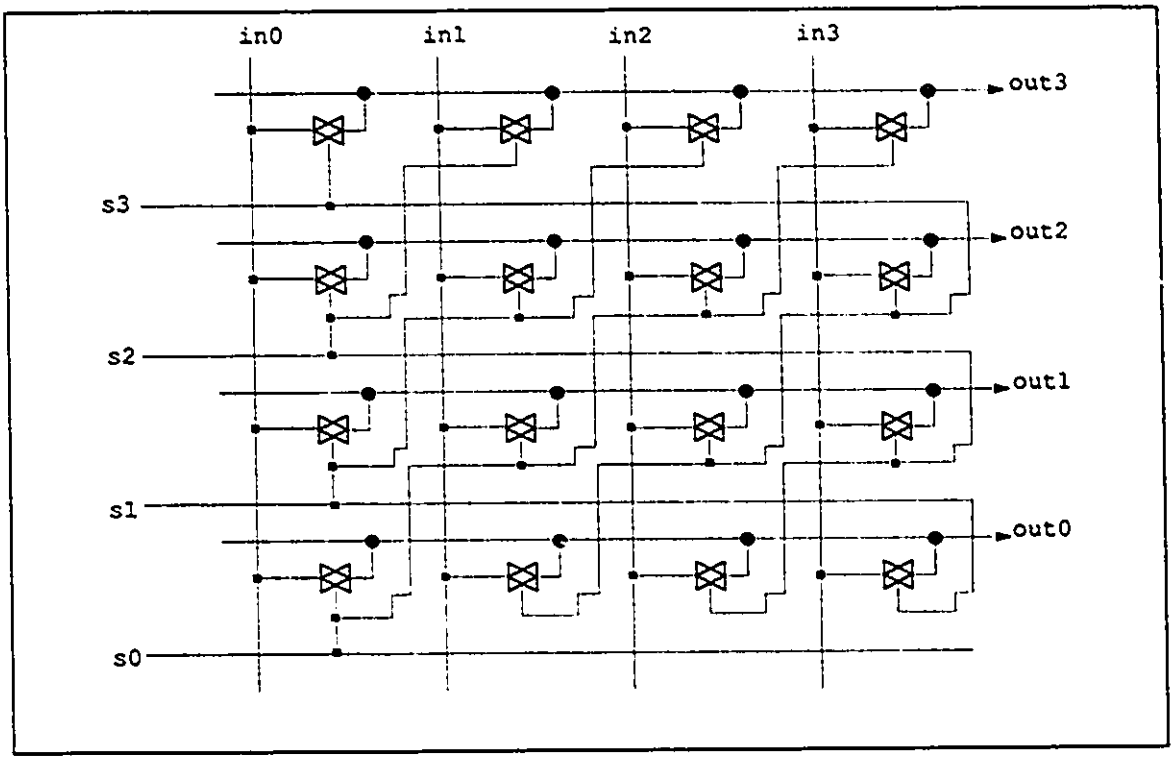



Figure 6.5.1. A four input barrel shifter implementation using transmission gates

This is not the only way to derive the barrel shifter with our transformation rules. In fact it could have been derived without the need for machine assistance, using the multiplexer rules.

## 6.6 Concluding comments

The most comprehensive work on formal modelling of transistors (in a logical framework) has been undertaken by Winskel [137]. C.A.R. Hoare's theory of MOS transistors [54] simplified and

[11] In this particular case, the only kind of don't cares that we have are input ones which are shared by all outputs. On the other hand, in case of output don't cares, it makes sense to have different designer's constraints for each individual output because this would result in more relaxed specifications.

recast Winskel's work as a design calculus for CMOS transistors. Our primary motivation was the formalization of Hoare's work within our ASDL framework. This means that directionality of transistor networks is imposed rather than derived. Although this may seem an odd choice for systems that are inherently bidirectional, it is not a prohibitive one for our intended use (synthesis); it simply means that a number of correctly functioning CMOS networks cannot be derived using our calculus. Since the number of truly bidirectional transistors in typical designs is small [35], this loss of generality is not as harmful as one might think. Some truly bidirectional CMOS circuits, like the bridge networks described in [24], cannot be analyzed using our model![12] The question, then is, if this problem can be attacked within the more general system semantics framework.

The answer is a qualified yes. It is possible to define semantic functions that take into account the fact that drain and source are interchangeable in a MOS transistor. In fact the work of [59], which describes "applicative" transistor models, can be readily extended with the drive and validity conditions. The problem is that these models adopt *object semantics*, that is, we have to assign unique labels to every transistor involved, which defeats one of the purposes of the applicative formalism, namely compositionality. Even if we define "appropriate" surface syntax to hide this problem as in Glass [103], the semantic functions will not be defined in a compositional manner. The incorporation of logic variables is a possible solution which would, nevertheless, complicate the semantics of our otherwise applicative framework. This is another indication that we have a long way to go in order to close the gap between hardware design languages and hardware description languages.

Finally, we mention the fact that another method for synthesizing combinational pass transistor networks using "incomplete" transmission gates, i.e. gates composed of either P or N-transistors,

---

[12] On the other hand, bridge networks introduce the possibility of sneak paths whose detection is known to be a co-NP hard problem.

without degrading the output signals is given in [105]. Their method, like ours, has been implemented in a software system, nevertheless it is not based in a formal transistor model and it cannot handle don't care values.

# Chapter 7 COMPUTER BASED TOOLS

The complete language and the semantic models described in chapter 3 have been implemented by the author, with the exception of isomorphic and user defined data types. In this chapter the software tools that have been written in order to test the ideas of the thesis are discussed.

## 7.2 Analysis and elaboration of ASDL definitions

Before we start analyzing a "module" containing a set of ASDL definitions, the C preprocessor is used in order to resolve included files. A default "prelude" containing the combinators defined in chapter 4 is also imported.

The first step in analyzing ASDL definitions is parsing. It has been written with the help of the Functional Parser Generator (FPG) [126], a system that takes *yacc*-like syntax specifications as input and produces LR(1) parsers in Lazy ML (LML) [2]. Unlike yacc, FPG accepts both inherited and synthesized attributes. Both FPG and LML are available in the public domain and this was instrumental in the decision to use them, along with the fact that they support a purely functional-lazy style of programming. The FPG specification of the current ASDL implementation is shown in appendix A. The lexical scanner is hand-coded and its major feature is the pure functional treatment of the *offside rule* (see Miranda™ manual, section 12).

The second step in analyzing ASDL definitions is scope analysis and flattening of local definitions. These are performed according to the rules of sections 3.2 and 3.3.1. Every identifier is annotated with its scope level (depth of local clause) and the number of the source equation in which it has been defined and, subsequently, all local definitions are moved leftward. Also, atoms and bus resolution functions defined in user supplied libraries are identified during this step.

The third step in analyzing ASDL definitions is typechecking. Our typechecker is a verbatim copy of the program appearing in the book by S.P. Jones [67], with additions to handle pattern-matching and inductive variables.

The next phase is elaboration. Not all system definitions have an interpretation as finite netlists (`Comp_def` in the wirelist intermediate language); keeping with the spirit of the language, we would like this fact to be inferred rather than explicitly indicated in the ASDL text. An arbitrary decision has been taken to partition system definitions into two sets according to their type signatures: first, **proper system definitions**, which have as signature a function type from a nested product of basetype(s) to a nested product and/or list of basetype(s); and, second, **system combinators**, which include all other system definitions (inductive definitions are part of them)[1]. During elaboration the right-hand-side of each proper system definition is "expanded" ($\beta$—reduced) using the definition of system combinators as rewrite rules, until no more rule applies. Since patterns are exhaustive and all elements of ASDL data types are finite, this process will eventually terminate. In addition to this, all function applications are "unfolded" with their signatures fully exposed to the level of basetypes. For example, the definition of the full-adder described in chapter 3:

```
fa (cin,x) => (or2 (tc1,tc2),sum)
              where
              (tc1,ts1) <= ha x
              (tc2,sum) <= ha (ts1,cin)
```

---

[1] This separation is arbitrary in the sense that a system with type signature `Wire->Wire->Wire` could be regarded as isomorphic to one with signature `(Wire,Wire)->Wire` (see [107]). Curried system definitions are semantically problematic, though, when partial applications are involved.

is unfolded into:

```
fa (cin,(a,b)) => (cout,sum)
                  where
                  cout <= or2 (tc1,tc2)
                  (tc1,ts1) <= ha (a,b)
                  (tc2,sum) <= ha (ts1,cin)
```

As a result of elaboration, all proper system definitions are free from occurrences of system combinators. They can be nested but cannot be (mutually) recursive, i.e. they are definitions in ASDL kernel form.

## 7.3 Extracting structure and behavior

ASDL kernel definitions are translated to the wirelist intermediate form using the method outlined in section 3.5. By default, this translation produces hierarchical wirelists, but the option to extract flat wirelists is also available. Since many VLSI tools, for example EDIF, cannot work if a subsystem is used before being defined, a topological sort of the resulting set of system definitions is also performed during this step. The great majority of today's hardware description languages are first-order and the wirelist intermediate form has been designed with this in mind. Therefore it is relatively straightforward to convert to any of the popular formats used by the VLSI community. As an example, appendix H contains the core Miranda™ module of a translator from the wirelist intermediate form to VHDL. This is the recommended route in order to import an ASDL description into the CADENCE™ design system. A translator to Queen's University Silicon Assembler for the ELECTRIC design system is also available.

Since isomorphic types are not provided in the current implementation of ASDL, the designer is offered the option to bypass the recursion restriction in inductive definitions. As a consequence, the effect of the data type abstraction can be "simulated" by inserting the type coercions at the appropriate places in the source text. Naturally, the responsibility of ensuring termination is now

delegated to the designer. As an example, we present the translation of the main example of chapter 5, the parallel prefix, instantiated with addition in place of the generic operator:

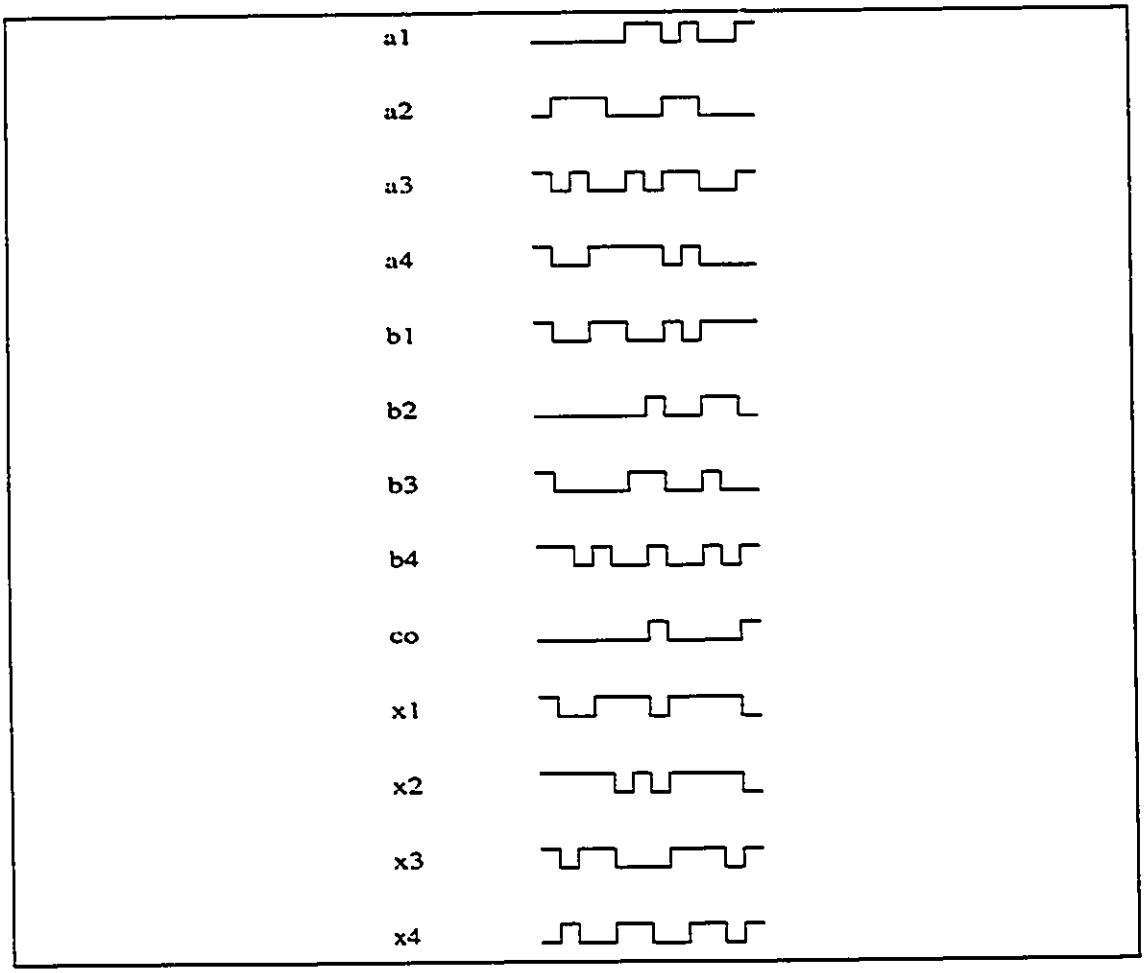
```
scanlt fxy e t => scanlti fxy e t e

scanlti fxy e []  d => ([],e)
scanlti fxy e [a] d => ([d],a)
scanlti fxy e xs  d
=> (ls ++ rs,u)
   where
   (u,(lrep,rrep)) <= fxy (d,(l,r))
   (ls,l) <= scanlti fxy e (take ((length xs) / 2) xs) lrep
   (rs,r) <= scanlti fxy e (drop ((length xs) / 2) xs) rrep

node (d,(l,r)) => (u,(d,r0))
                  where
                  u  <= add (l,r)
                  r0 <= add (d,l)
scant16 (a,(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,
            x12,x13,x14,x15,x16))
 => scanlt node a [x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,
                   x12,x13,x14,x15,x16]
```

In appendix I, the wirelist form and the VHDL description generated from it are given.

Simulation of ASDL descriptions is performed by translating them into Miranda™ scripts. Since ASDL kernel form is in essence the abstract syntax of a functional language, this translation is simply an unparsing into concrete Miranda code. Behavioral libraries, corresponding to the atoms and bus resolution functions of the computational basis, are linked with the Miranda model. The default library is that of the stream behavioral model. A user can create her (his) library by: *(i)* specifying the signatures of the atoms and bus resolution functions in a single file and *(ii)* developing the corresponding behavioral models (in Miranda) in separate files. In case that the atoms involved are combinational, only the instantaneous behavioral models have to be provided. The syntax of library specifications is given in appendix A. The output and the test vectors used in a simulation run can be visualized by a Postscript™ waveform viewer. Figure 7.3.1 depicts the input-output waveforms of the four bit carry-lookahead adder with inputs the sequence of pairs

(in integer form) `[(3,11),(4,1),(6,0),(5,9),(1,8),(11,2),(9,7),(6,8),(15,0),(0,15),(0,12),(10,9)]`:

Figure 7.3.1. Input-output waveforms for carry-lookahead adder




## 7.5 The CMOS synthesizer

The algorithms described in sections 6.4 and 6.5 have been implemented in Haskell. The core module of the algorithm that synthesizes optimized CMOS circuits from incomplete specifications is given in appendix G. As an example, the unoptimized mux4 (without applying the

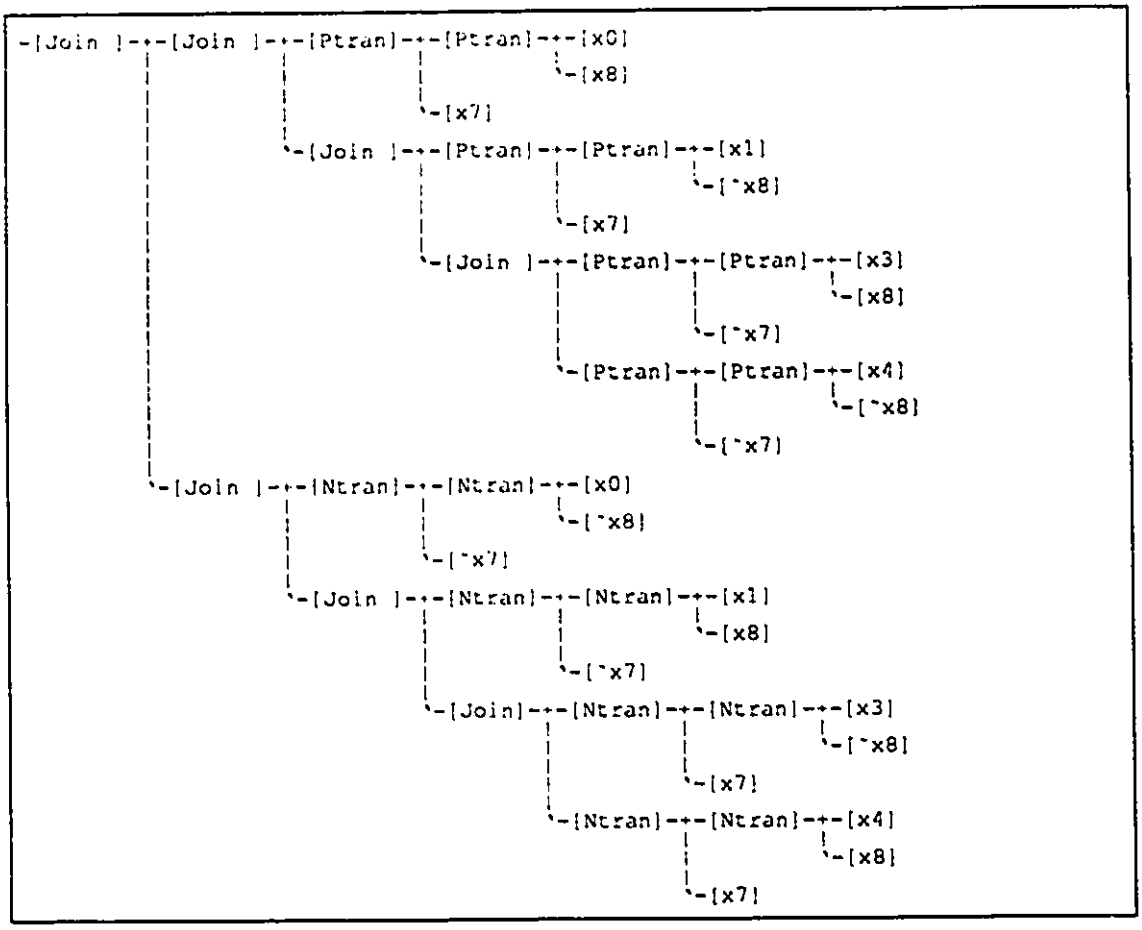
conditional transformation rules *ntranRseries* and *ptranRseries*) is unparsed into the following ASDL definition:

```
mux4 (x7,x8,x0,x1,x3,x4)
=> join (join (ptran (x7,ptran (x8,ptran (x0_bar,pwr))),
   join (ptran (x7,ptran (x8_bar,ptran (x1_bar,pwr))),
   join (ptran (x7_bar,ptran (x8,ptran (x3_bar,pwr))),
         ptran (x7_bar,ptran (x8_bar,ptran (x4_bar,pwr))))))),
   join (ntran (x7_bar,ntran (x8_bar,ntran (x0_bar,gnd))),
   join (ntran (x7_bar,ntran (x8,ntran (x1_bar,gnd))),
   join (ntran (x7,ntran (x8_bar,ntran (x3_bar,gnd))),
         ntran (x7,ntran (x8,ntran (x4_bar,gnd))))))
      where // use inverters to negate primary inputs
      x0_bar <= join (ptran (x0,pwr), ntran (x0,gnd))
      x8_bar <= join (ptran (x8,pwr), ntran (x8,gnd))
      x1_bar <= join (ptran (x1,pwr), ntran (x1,gnd))
      x7_bar <= join (ptran (x7,pwr), ntran (x7,gnd))
      x3_bar <= join (ptran (x3,pwr), ntran (x3,gnd))
      x4_bar <= join (ptran (x4,pwr), ntran (x4,gnd))
```

Applying the two additional rules, we obtain:

```
mux4opt (x7,x8,x0,x1,x3,x4)
 => join (join (ptran (x7,ptran (x8,x0)),
    join (ptran (x7,ptran (x8_bar,x1)),
    join (ptran (x7_bar,ptran (x8,x3)),
          ptran (x7_bar,ptran (x8_bar,x4))))),
    join (ntran (x7_bar,ntran (x8_bar,x0)),
    join (ntran (x7_bar,ntran (x8,x1)),
    join (ntran (x7,ntran (x8_bar,x3)),
          ntran (x7,ntran (x8,x4))))))
    where
    x8_bar <= join (ptran (x8,pwr), ntran (x8,gnd))
    x7_bar <= join (ptran (x7,pwr), ntran (x7,gnd))
```

A rudimentary floorplan of all synthesized circuits can be extracted, for example the `mux4opt` described above has the form:




where the direction of the dataflow is from right to left and the following convention is used for drawing transistors:


In addition to the above, a simple module for factorization of common subexpressions in boolean specifications has been written. The performance of the synthesizer is satisfactory, for example the most complex of the examples presented in chapter 6, the four output barrel shifter, is synthesized in 482,207 reductions which take less than 30 seconds on a Sparcstation 1 with 8MBs of memory.

# Chapter 8 CONCLUSION

In this chapter, we review the research that had a major influence on this investigation, the most important results of the thesis and the prospects for future research.

## 8.1 Related work

Owing to the great impact of John Backus's Turing award talk [5], a number of hardware description methodologies based on FP were proposed during the first half of the eighties.

The Functional Hardware Description Language (FHDL) [95] and $\nu$FP [38] were developed at UCLA for the multilevel specification, analysis and synthesis of digital systems. They have been mainly used to describe implementations of algorithms for special purpose arithmetic using cell iterative arrays. The major contribution of the UCLA group is the development of algorithms [111], [110] for extracting the topology (global routing and planar graphs) from $\nu$FP specifications and, subsequently, actual layout using a form of vertical compaction. The limitations of these algorithms are that no feedback is allowed and that data flow has to be unidirectional.

Array FP (aFP) [82] has been designed as a language for describing programmable systolic systems. The major innovation of aFP is that the user can specify the scheduling of the input data. Related to this is the work reported in [125] where as-soon-as-possible resource allocation and scheduling algorithms are used in order to extract architectures from behavioral specifications expressed in a subset of FP. For another approach to imposing temporal constraints in order to control the behavior of applicative systems see [40]. The claimed advantage of all these approaches is that a rapid evaluation of space-time trade-offs is made possible.

$\mu$FP [112], [114] is an extension of FP with an additional combining form that allows the description of state, and the re-interpretation of the behavioral semantics of all FP constructs in the stream model, in line with our approach described in section 3.6.2. Reportedly [86], [84], tools have been developed at Oxford in order to support development and extract performance and other alternative interpretations of circuits described in the $\mu$FP framework. The major contribution of $\mu$FP is its associated algebra [113] which treats combinational and sequential systems uniformly. The major deficiencies of the $\mu$FP approach are the lack of data typing facilities and its inability to describe systems with bidirectional data flow in an effective way. Much of our initial motivation for this thesis grew out in our attempt to overcome these limitations.

In order to overcome the second of these deficiencies, the author of the original $\mu$FP language, M. Sheeran, along with G. Jones have developed RUBY [119], [63], [117], [64], a language with binary-relational combining forms but functional primitives. The fact that the primitives (atoms) of RUBY are unidirectional means that its expressive power is similar to that of ASDL, i.e. only circuits that have a behavioral interpretation which is a total function from its input to its output can be described. The claimed advantage of RUBY is that, owing to the symmetry of the relational notation, fewer steps are needed for the derivation of regular circuits (in particular). It is true, also, that the geometric interpretation of RUBY's combining forms is more natural than that of $\mu$FP. More recent work [66] shows how types can be modelled as partial equivalence relations with the consequence that data and a form of timing abstractions [65] (bit-serial) can be expressed within the language. Finally, we note that a small subset of the language (formal RUBY) has been "implemented" in a theorem prover [108].

In our opinion, there is a serious drawback in the RUBY approach[1]: not all syntactically correct descriptions definable in it make sense as real circuits. A separate proof is required in

---

[1] Another drawback of relational notations with respect to applicative ones is that properties of the symbols involved, i.e. associativity, cannot be expressed in a succinct way.

order to show that the description is "implementable", i.e. there exists an assignment of directions for all wires in the circuit which is consistent with that of the primitives [118, section 15]. This is a high price to pay for an approach that does not even allow the description of truly bidirectional systems! We believe that reasoning about the direction of data flow should be "factored" into the design process as we have demonstrated with our Temporal Attribute Grammars approach described in chapter 5. In addition to this, animation of specifications is much more natural in the applicative framework.

S.D. Johnson argues very convincingly in his thesis [56] that the applicative notation is a fitting basis for digital design based on the fact that the means of abstraction (functionality) are the same in both applicative programming and digital design. He has also demonstrated that systems of recursion equations in a first order applicative language can be interpreted as specifications of schematics. Although his original work was more compiler oriented (derivation of language interpreters from continuation semantics specifications), his more recent research [59], [57] has a more algebraic flavor to it. HYDRA [43] is a language closely based on Johnson's work.

STREAM is another language developed by Kloos [72] for the description of loosely-coupled digital systems. A simple procedural language with strict semantics is also introduced as an interface between hardware and software, but perhaps his most important contribution is the study of hazards in terms of fixpoint theory. Like the FP derivatives mentioned above, STREAM lacks data typing facilities.

ELLA [100] is a commercial hardware description language developed by the UK MoD with applicative semantics, very similar to ASDL. It is strictly typed and is powerful enough to describe a high-level implementation in a form that can be interpreted as a behavioral specification. EllaX [135] extends the temporal model of ELLA and provides for output don't cares and unspecified conditions.

STRICT [26] is another strictly typed hardware description language which describes systems

in terms of buses (transmitters of data of a specific type) and blocks (data transformers or representation changers).

$F^2$ [28] is a functional formalism used to describe synchronous systems as multiple-output first-class recursive functions. Recursion is allowed both over structure and over time. Recursion over structure allows the description of iterative systems and recursion over time allows the description of sequential behavior in the form of difference equations. Examples of systolic array descriptions are given in [27].

HIFI [1] is a system combining Communicating Sequential Processes (CSP) and Applicative State Transition (AST) systems. The CSP communication model is used to express concurrency and to make verification possible while the atomic processes are refined to AST systems. The major use of HIFI is signal processing.

Finally, Glass [30], [31] is a system description language that, in a way, served as a yardstick for ASDL. It is more general than ASDL in that it can describe analog systems and adirectional systems [20] but it is first order, lacks inductive definitions and its type system is very limited. The Glass framework has been developed by R. Boute [15], [19] and its use in transformational design is demonstrated in [33] and [89].

## 8.2 Important results

We have demonstrated that interesting technology transfer from applicative programming to VLSI design has great potential and deserves further attention from the researchers in the field. Although, we feel, the large standardization effort that culminated with the design of VHDL is not likely to be repeated in the near future, it is our hope that the thesis will stimulate new interest in the construction of environments that support applicative techniques in hardware design.

In the following, we list the specific results that are relevant to the two aspects of the thesis:

**I. Results that support the claim that VLSI design can be regarded as a form of applicative programming**

a. Previous research, reviewed in the previous section, has shown that particular aspects of the VLSI design process can be regarded as specific forms of applicative programming. More specifically, M. Sheeran has shown that FP's predefined higher order combinators can be interpreted as specifications of regular interconnection patterns with nice topological properties and S.D. Johnson has shown that the applicative notation is a fitting basis for digital design because the means of abstraction (functionality) are the same in both domains. Our work has generalized both of the above approaches in a unified framework and, in addition, has provided a facility for the designer to define her (his) own combinators.

b. We are the first to show that one does not need a relational framework, as in [62], in order to reason about regular circuits with bidirectional data flow. This is important because the problems related with the relational approach, i.e. realizability of specifications, difficulties in design animation, educating designers with new abstractions of the design process, can be avoided.

c. We have shown without doubt that restricting recursion to "well-founded recursion" within a higher order applicative framework does not appear to reduce the expressive power of the notation in describing VLSI architectures. The advantage of this approach is that it prevents design errors, i.e. all circuits described in this notation have finite layout. Another benefit is that our framework can be embedded more easily in modern formal systems such as Lambda [49], [46] or Veritas+ [52].

d. We are the first to provide a model of the detailed timing behavior based on fixpoint theory. This is important because it enables the application of the same techniques used throughout the thesis in reasoning about the timing behavior.

**II. Results that support the claim that formal methods that have been developed for the applicative programming paradigm can be adapted and used to advantage in VLSI design**

Previous research has demonstrated that some formal techniques from the field of applicative programming can be used to advantage in VLSI design. More specifically, S.D. Johnson has shown how the transformation of data types technique can be used to obtain iterative realizations of circuits and M. Sheeran has shown how FP's algebra of programs can be "lifted" to the domain of circuits. Our investigation has generalized Sheeran's work by demonstrating how the more modern Bird and Meertens algebra can be transferred to the circuit domain. Also, we have illustrated how techniques from the attribute grammar field can be used in the design of circuits. More specifically:

a. We have clearly demonstrated that Bird and Meertens's approach to transformational programming [12] can be adapted for use in VLSI design.

b. We have stated and proved new fusion laws for list and tree paramorphisms. Fusion laws for homomorphisms are special cases of these.

c. We have shown how techniques for coupling attribute grammars (which are applicative formalisms) can be used in VLSI design. We are also the first to generalize Leiserson's retiming lemma in retiming synchronous circuits described as attribute grammars.

d. We have demonstrated how abstraction constraints can be expressed in the applicative framework by using CMOS circuits as the test case.

In addition to the results above which support the thesis, the fact that it was possible to construct the software tools discussed in chapter 7 within the constraints of a Ph.D. work, lends support to the growing recognition that the applicative programming paradigm is appropriate for VLSI design.

## 8.3 Prospects for future research

The stream and the timing behavioral models, which have been introduced in terms of fixpoint theory in sections 3.6.2 and3.6.3, can be re-expressed in terms of "terminal algebras" [88]. Whereas reductions correspond to recursive homomorphisms on finite lists, accumulations now correspond to recursive homomorphisms on infinite lists (streams). The immediate benefit from this approach is that the "principle of duality" [ibid, section 4] can be now applied on the fusion laws making it possible to reason about timing abstractions, for example serialization or relating two different levels of timing, in the same algebraic framework as the one used in this report for reasoning about ASDL.

Another direction for further research is the extension of ASDL's type system. It seems relatively easy to extend the type checking system so that cardinality restrictions on finite lists can be imposed (see [60]), as in the following example:

```
zipwith :: ((a,b)->c) -> [a]^n -> [b]^n -> [c]^n
```

which expresses the constraint that the two input lists must be of the same length which is also the length of the result list. Constraints like that are commonplace in describing architectures for fixed precision arithmetic. Low level hardware description, requires even more general constraints. For example, MOS transistors could be given a more "accurate" type if the full power of dependent types and subtypes [52] was used to model the drive and validity conditions discussed in chapter 6. Nevertheless, the decidability of such type systems is an open research question.

This brings us to the following question: is it possible to refine types to the level of behavioral specifications and then use the "programs from proofs" paradigm [37], to extract circuits from types? Basin, in his thesis [8, pp. 92–98], argues that this simple-minded analogy is not always valid in that there is no assurance that programs extracted from proofs are always realizable as circuits and that sometimes different "circuits" are extracted for different input values rather than a single circuit that works for all inputs. One solution could be to define the type of ASDL

expressions as a recursive type in type theory [4] and then employ the behavioral semantic functions of section 3.6 in order to assign meaning to circuits encoded in ASDL. This is probably the best method of incorporating ASDL into a formal system. We plan to investigate this approach by using an industrial strength example (a modern microprocessor) as a case study in order to test the modularity and assess the user friendliness of the notation as well.

The extension of ASDL to the direction of systems that cannot be usefully abstracted in terms of a causal input-output relation will make the description of adirectional systems, such as the CMOS transistor networks discussed in chapter 6, more symmetrical. Rather than abandon unidirectional models, the sigma calculus [20] is a framework that allows their co-existence with adirectional ones.

Finally, the construction of a user-friendly interface using "structured" schematics in a way similar to the commercial system Lambda [90], [46], will make formal methods more appealing to practicing engineers.

# BIBLIOGRAPHY


[1] J. Annevelink and P. Dewilde. HIFI: A Functional Design System for VLSI. In *International Conference on Systolic Arrays*. Computer Society Press, 1988.

[2] Lennart Augustsson and Thomas Johnsson. *Lazy ML user's manual*, 0.998.2 edition, 1992.

[3] R.C. Backhouse. An exploration of the Bird–Meertens Formalism. Technical Report Computing Science Notes CS 8810, Department of Mathematics and Computing Science, University of Groningen, 1988.

[4] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Eric Saaman. Do-it-yourself Type Theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[5] John Backus. Can Programming Be Liberated from the von Neuman Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8), 1978.

[6] John Backus, J.H. Williams, and E.L. Wimmers. An introduction to the Programming Language FL. In David A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin, Year of Programming Series. Addison-Wesley, 1990.

[7] David A. Basin, Geoffrey M. Brown, and Miriam E. Leeser. Formally verified synthesis of combinational CMOS circuits. *Integration, the VLSI journal*, 11:235–250, 1991.

[8] David Alan Basin. *Building problem-solving environments in constructive type theory*. PhD thesis, Cornell University, 1990.

[9] Bernd Becker, Gunter Hotz, Reiner Kolla, Paul Molitor, and Hans-Georg Osthof. Hierarchical Design based on a Calculus of Nets. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*. IEEE press, 1987.

[10] Richard S. Bird. A Calculus for Program Derivation. In *Research Topics in Functional Programming*, The University of Texas Year of Programming Series, chapter 11. Addison-Wesley Publishing Company, Inc., 1990.

[11] R.S. Bird. An introduction to the Theory of Lists. volume 36 of *NATO Advanced Studies Institute Series F*, pages 3–42. Springer-Verlag, 1987.

[12] R.S. Bird. Lectures on Constructive Functional Programming. In Manfred Broy, editor, *Constructive Methods in Computer Science*. Springer-Verlag, 1989.

[13] G. Birtwistle and P.A. Subrahmanyam, editors. *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.

[14] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. PhD thesis, MIT Press, 1990.

[15] Raymond T. Boute. System Semantics and Formal Circuit Description. *Transactions on Circuits and Systems*, CAS-33(12):1219–1231, 1986.

[16] Raymond T. Boute. On the formal description of non-computational objects. In *IFIP WG 10.1 Workshop on Concepts and Characteristics of Declarative Systems*, 1988.

[17] Raymond T. Boute. System Semantics: Principles, Applications, and Implementation. *ACM Transactions on Programming Languages and Systems*, 10(1), 1988.

[18] Raymond T. Boute. Representational and Denotational Semantics of Digital Systems. *IEEE Transactions on Computers*, 38(7):986–999, 1989.

[19] Raymond T. Boute. Syntactic and Semantic Aspects of Formal System Description. In *Microprocessing and Microprogramming*, volume 27. North-Holland, 1989.

[20] Raymond T. Boute. The Sigma Calculus: Scoping and Substitution in Formal Description of Systems that are Not Unidirectional. Unpublished manuscript, University of Nijmegen, The Netherlands, 1989.

[21] Raymond T. Boute. Declarative languages - still a long way to go. In *Computer Hardware Description Languages and their Application*. Elsevier Science Publishers, 1991.

[22] R.K. Brayton, G.D. Hachtell, and A.L. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *IEEE Proceedings*, 78(2), 1990.

[23] Manfred Broy. Predicative specifications for functional programs describing communicating networks. *Information Processing Letters*, 25, 197.

[24] J.A. Brzozowski and M. Yoeli. Combinational static CMOS networks. *INTEGRATION, the VLSI Journal*, 5, 1987.

[25] R. M. Burstall. Inductively defined functions in Functional Programming Languages. *Journal of Computer and System Sciences*, 34, 1987.

[26] R.H. Campbell, A.M. Koelmans, and M.R. McLauchlan. STRICT: a design language for strongly typed recursive integrated circuits. *IEE Proceedings, Parts E and I*, 132(2), 1985.

[27] Paolo Camurati, Tiziana Margaria, and Paolo Prinetto. Systolic array description in $F^2$. In *Microprocessing and Microprogramming*, volume 27, 1989.

[28] Paolo Camurati, Tiziana Margaria, and Paolo Prinetto. VLSI functional descriptions in $F^2$. In D.A. Edwards, editor, *Design Methodologies for VLSI and Computer Architecture*. IFIP, Elsevier (North-Holland), 1989.

[29] Luca Cardelli and Peter Wegner. On understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4), 1986.

[30] Catholic University of Nijmegen. *Glass: A systems description language and its environment: Introduction and User manuals, (vol. 1)*, 3rd edition, 1992.

[31] Catholic University of Nijmegen. *Glass: A systems description language and its environment: Language Reference manual, (vol. 2)*, 3rd edition, 1992.

[32] Kung Chen, Paul Hudak, and Martin Odersky. Parametric Type Classes. In *Conference on Lisp and Functional Programming*. ACM, 1992.

[33] L. Claesen, R.T. Boute, J.DE Man, and M.Seutter. Application of System Semantics to VLSI for the Transformational Design of a Parameterized Booth Multiplier Module - a case study. In *Microprocessing and Microprogramming*, volume 27. North-Holland Publishing Company, 1989.

[34] Edmund Clarke and Yulin Feng. Escher — A Geometrical Layout System for Recursively Defined Circuits. Technical Report CMU-CS-85-150, Dept. of Computer Science, Carnegie-Mellon University, July 1985.

[35] W.F. Clocksin. Logic Programming and Digital Circuit Analysis. *Journal of Logic Programming*, 10, 1987.

[36] M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.

[37] R.L Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangraden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[38] Patel D., M. Schlag, and M. Ercegovac. vFP: An environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms. In *Functional Programming and Computer Architecture*, number 201 in Lecture Notes in Computer Science. Springer-Verlag, 1985.

[39] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*. ACM, 1982.

[40] John Darlington and Lyndon While. Controlling the Behavior of Functional Language Systems. In *Functional Programming and Computer Architecture 1987*, number 274 in Lecture Notes in Computer Science, pages 278–300. Springer-Verlag, 1987.

[41] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*. Number 323 in Lecture Notes in Computer Science. Springer-Verlag, 1988.

[42] Doaitse Swierstra and Harald Vogt. Higher Order Attribute Grammars. In *Attribute Grammars, Applications and Systems*, number 545 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[43] John T. O' Donnel. Hardware Description with Recursion Equations. In M.R. Barbacci and C.J. Koomen, editors, *Computer Hardware Description Languages and their Applications*. IFIP, Elsevier (North-Holland), 1987.

[44] Piloty et al. *CONLAN Report*. Number 151 in Lecture Notes in Computer Science. Springer-Verlag, 1983.

[45] Hans Eveking. The application of CONLAN assertions to the correct description of hardware. In M. Bruer and R. Hartenstein, editors, *Computer Hardware Description Languages*. North-Holland, 1981.

[46] Simon Finn, M.P. Fourman, M. Francis, and R. Harris. Formal System Design - Interactive Synthesis based on Computer-Assisted Formal Reasoning. In Luc J.M. Claesen, editor, *Formal VLSI Specification and Synthesis, VLSI Design Methods-I*. North-Holland, 1990.

[47] John P. Fishburn. A Depth-Decreasing Heuristic for Combinational Logic; or How to Convert A Ripple-Carry Adder into a Carry-Lookahead Adder or Anything In-Between. In *27th ACM/IEEE Design Automation Conference*. ACM-SIGDA/IEEE Computer Society-DATC, 1990.

[48] Wai Fong. Use of the Functional Programming paradigm in VLSI Specification and Design. Master's thesis, University of Windsor, School of Computer Science, 1991.

[49] Michael P. Fourman and Eleanor M. Mayger. Formally based system design - Interactive Hardware Scheduling. In *Proceedings of VLSI '89, Munich FRG*, pages 101–112. 1989.

[50] M.P. Fourman. Formal System Design. In Staunstrup [122], chapter 5, pages 191–236.

[51] R.A. Frost. Constructing Programs as Executable Attribute Grammars. *Computer Journal*, to appear.

[52] F. Keith Hanna, Neil Daeche, and Mark Longley. Specification and Verification Using Dependent Types. *IEEE Transactions on Software Engineering*, 16(9):949–964, September 1990.

[53] John Herbert. Formal Reasoning about the timing and function of basic memory devices. In Luc J.M. Claesen, editor, *Formal VLSI Specification and Synthesis, VLSI Design Methods-II*. North-Holland, 1990.

[54] C.A.R. Hoare. A Theory for the Derivation of C-MOS Circuit Designs. In *Beauty is Our Business, A Birthday Salute to E.W. Dijkstra*, pages 194–205. Springer-Verlag, 1991.

[55] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the Programming Language Haskell, Version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[56] S.D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. PhD thesis, Indiana University, 1983.

[57] S.D. Johnson. Manipulating Logical Organization with System Factorizations. In Leeser and Brown [77]. Proceedings of Mathematical Sciences Institute Workshop, Cornell University, July 1989.

[58] S.D. Johnson, Baskar Bose, and C.D. Boyer. A Tactical Framework for Hardware Design. In Birtwistle and Subrahmanyam [13].

[59] S.D. Johnson and C.D. Boyer. Modelling Transistors Applicatively. In Milne [96], pages 397–420.

[60] Geraint Jones. Deriving the Fast Fourier algorithm by calculation. In Kei Davis and John Hughes, editors, *Second Glasgow Workshop on Functional Programming*. Springer-Verlag, 1990.

[61] Geraint Jones and Mary Sheeran. Timeless Truths about Sequential Circuits. In Stuart K. Tewksbury, Bradley W. Dickinson, and Stuart C. Schwartz, editors, *Concurrent computations: Algorithms, Architecture, and Technology*, pages 245–260. Plenum Press, 1987.

[62] Geraint Jones and Mary Sheeran. Circuit Design in RUBY. In Staunstrup [122], chapter 1, pages 13–70.

[63] Geraint Jones and Mary Sheeran, editors. *Designing Correct Circuits*. Workshops in Computing. Springer-Verlag, 1990.

[64] Geraint Jones and Mary Sheeran. Relations and Refinement in circuit design. Technical Report PRG-TR-13-90, Oxford University Computing Laboratory, 1990.

[65] Geraint Jones and Mary Sheeran. Deriving bit-serial circuits in RUBY. Technical Report PRG-TR-3-91, Oxford University Computing Laboratory, 1991.

[66] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in RUBY. Technical Report PRG-TR-1-92, Oxford University Computing Laboratory, 1992.

[67] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[68] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages*. Prentice-Hall, 1992.

[69] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP '74*, 1974.

[70] Gilles Kahn and David B. MacQueen. Coroutines and Networks of Parallel Processes. In *IFIP Conference*. North-Holland, 1977.

[71] Paul Kelly. *Functional Programming for Loosely-coupled Multiprocessors*. PhD thesis, Imperial College of Science and Technology, 1989.

[72] Carlos Delgado Kloos. *Semantics of Digital Circuits*. PhD thesis, Institut fur Informatic der Technischen Universitat Munchen, 1987. Published by Springer-Verlag, LNCS 285.

[73] Carlos Delgado Kloos. Transformational Development of Circuit descriptions for Binary Adders. In M. Broy and M. Wirsing, editors, *Methods of Programming*, number 544 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[74] Carlos Delgado Kloos and Walter Dosch. Efficient Circuits as Implementations of Non-Strict Functions. In Jones and Sheeran [63].

[75] S. Y. Kung. *VLSI Array Processors*. Prentice-Hall, 1988.

[76] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of the ACM*, 27(4), 1980.

[77] M. Leeser and G. Brown, editors. *Hardware Specification, Verification and Synthesis: Mathematical Aspects*. Number 408 in Lecture Notes in Computer Science. Springer-Verlag, 1990. Proceedings of Mathematical Sciences Institute Workshop, Cornell University, July 1989.

[78] Charles E. Leiserson and James B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, 1983.

[79] Charles Eric Leiserson. *Area-Efficient VLSI Computation*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1981.

[80] Karin Lemmert. Data path descriptions. In R.W. Hartenstein, editor, *Hardware Description Languages*. North-Holland, 1987.

[81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall Inc., 1981.

[82] Yen-Chun Lin and Feng-Ching Lin. The use of aFP to Design Regular Array Algorithms. In *IEEE International Conference on Computer Languages*, 1988.

[83] Wayne Luk. *Parameterized design of regular processor arrays*. PhD thesis, Oxford University, PRG, 1988.

[84] Wayne Luk. Analyzing parameterized designs by non-standard interpretation. In S.Y. Kung, E.E. Swartzlander, A. Fortes, and K. Przytula, editors, *Application specific array processors*. IEEE Press, 1990.

[85] Wayne Luk. Specifying and developing regular heterogeneous designs. In Luc J.M. Claesen, editor, *Formal VLSI Specification and Synthesis. VLSI Design Methods-I*. North-Holland, 1990.

[86] Wayne Luk, Geraint Jones, and Mary Sheeran. Computer-based tools for regular array design. In *International Conference on Systolic Arrays*, 1989.

[87] Grant Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

[88] Grant Malcolm. Data structures and Program Transformation. *Science of Computer Programming*, 14:255–279, 1990.

[89] Jozef De Man. Transformational Design: A Case Study. In Luc J.M. Claesen, editor, *Formal VLSI Specification and Synthesis. VLSI Design Methods-I*. North-Holland, 1990.

[90] E.M. Mayger, M.D. Francis, R.L. Harris, G. Musgrave, and M.P. Fourman. DIALOG - Linking Formal Proof to the Design Environment. Technical report, Abstract Hardware Ltd, 1989.

[91] Lambert Meertens. Constructing a calculus of programs. In *Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

[92] Eric Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In John Hughes, editor, *Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[93] Thomas F. Melham. Abstraction Mechanisms for Hardware Verification. In Birtwistle and Subrahmanyam [13].

[94] Paul Francis Mendler. *Inductive Definitions in Type Theory*. PhD thesis, Cornell University, 1987.

[95] F. Meshkinpour and M.D. Ercegovac. A Functional Language for Description and Design of Digital Systems: Sequential Constructs. In *22nd Design Automation Conference*, 1985.

[96] George J. Milne, editor. *IFIP WG 10.2 Working Conference on the Fusion of Hardware Design and Verification*. North-Holland, 1988.

[97] George J. Milne. Design for Verifiability. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, number 408 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

[98] Robert Milne. Design Transformation and Chip Planning. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. Elsevier (North-Holland), 1986. LTS.

[99] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–75, 1978.

[100] J.D. Morison, N.E. Peeling, and T.L. Thorp. The design rationale of ELLA, a Hardware design and description language. In C.J. Koomen and T. Moto-oka, editors, *Computer Hardware Description Languages and their Applications*, pages 303–320. IFIP, Elsevier (North-Holland), 1985.

[101] D.A. Musser, P. Narendam, and W.J. Premerlani. BIDS: A Method for Specifying and Verifying Bidirectional Hardware Devices. In Birtwistle and Subrahmanyam [13], chapter 6.

[102] Theodore S. Norvel and Eric C.R. Hehner. Logical Specifications for Functional Programs. Technical report, University of Toronto, Department of Computer Science, 1992.

[103] H. Oolman, M. Scutter, and C. van Reeuwijk. Glass, a language for analog and digital circuit description, and its environment. Number 27 in Microprocessors and Microprogramming, pages 267–271. North-Holland, 1989.

[104] Helmut A. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development*. Springer-Verlag, 1990.

[105] Corrado Pedron and Andre Stauffer. Analysis and Synthesis of Combinational Pass Transistor Circuits. *IEEE Transactions on CAD*, 7(7), 1988.

[106] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[107] Mikael Rittri. Using Types as Search Keys in Fuction Libraries. In *Functional Programming and Computer Architecture*. ACM Press, 1989.

[108] Lars Rossen. Formal RUBY. In Staunstrup [122].

[109] A. Sarkar, S. Bandyopadhyay, and G. A. Jullien. Layout driven logic synthesis. submitted for publication at Euro-DAC '93, 1993.

[110] Martine Schlag. The Planar Topology of Functional Programs. In *Functional Programming and Computer Architecture*, number 274 in Lecture Notes in Computer Science, 1987.

[111] Martine Denise Fr. Schlag. *Layout from a Topological Description*. PhD thesis, UCLA, 1986. UMI.

[112] Mary Sheeran. *$\mu$FP - An Algebraic VLSI Design Language*. PhD thesis, Oxford University — Computing Laboratory — Programming Research Group, 1983.

[113] Mary Sheeran. Designing Regular Array Architectures using Higher Order Functions. In *Functional Programming and Computer Architecture*, number 201 in Lecture Notes in Computer Science. Springer-Verlag, 1985.

[114] Mary Sheeran. Design and verification of regular synchronous circuits. *IEE Proceedings, Part E*, 133(5):295–304, 1986.

[115] Mary Sheeran. Retiming and Slowdown in RUBY. In Milne [96].

[116] Mary Sheeran. Categories for the Working Hardware Designer. In Leeser and Brown [77]. Proceedings of Mathematical Sciences Institute Workshop, Cornell University, July 1989.

[117] Mary Sheeran. Describing and reasoning about circuits using relations. In J.V. Tucker K. McEvoy, editor, *Theoretical Foundations of VLSI Design*, Cambridge Tracts in Theoretical Computer Science, pages 263–298. Cambridge University Press, 1990.

[118] Mary Sheeran. Describing Hardware algorithms in RUBY. In G. David, R.T. Boute, and B.D. Shriver, editors, *Declarative Systems*. Elsevier (North-Holland), 1990.

[119] Mary Sheeran and Geraint Jones. Relations + Higher Order Functions = Hardware Descriptions. Technical Report CSC/87/R1, University of Glasgow, Dept. of Computing Science, 1987.

[120] S. Singh. An application of non-standard interpretation: testability. In Luc J.M. Claesen, editor, *Formal VLSI Specification and Synthesis, VLSI Design Methods-II*. North-Holland, 1990.

[121] D.B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. Technical report, Department of Computing and Information Science, Queen's University, Kingston, Canada, 1992.

[122] Jørgen Staunstrup, editor. *Formal Methods for VLSI Design*. North-Holland, 1990.

[123] Masato Takeichi. Partial Parametrization Eliminates Multiple Traversals of Data Structures. *Acta Informatica*, 24:54–77, 1987.

[124] Satish R. Thatte. Coercive Type Isomorphism. In John Hughes, editor, *Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[125] P. Tsanakas, N. Alexandridis, and G. Papakonstantinou. An FP-based Design Methodology for Problem-oriented Architectures. *The Computer Journal*, 32(5):453–460, 1989.

[126] Goran O. Uddeborg. A Functional Parser Generator. Technical Report 43, Progarmming Methodology Group, Department of Computer Science, Chalmers University of Technology and University of Goteborg, SWEDEN, 1988.

[127] Marko van Eekelen, Halbe Huitema, Eric Nocker, Sjaak Smetsers, and Rinus Plasmeijer. *Concurrent Clean language manual*, 0.8 edition, 1992.

[128] Harald Vogt, Aswin van den Berg, and Arend Frelje. Rapid development of a program transformation system with attribute grammars and dynamic transformations. In *WAGA '90*, Lecture Notes in Computer Science. Springer-Verlag, 1990.

[129] William W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13:3–15, 1981.

[130] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*. ACM, 1987.

[131] Albert Ren Rui Wang. *Algorithms for multilevel logic optimization*. PhD thesis, University of California, Berkley, 1989.

[132] Daniel Weise. Constraints, Abstraction, and Verification. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, number 408 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

[133] Daniel Weise. Multilevel Verification of MOS Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(4):341–351, 1990.

[134] Neil Weste and Kamran Eshragarian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley, 1985.

[135] Gregory S. Whitecomb and A. Richard Newton. Abstract Data Types and High-Level Synthesis. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE Press, 1990.

[136] Glynn Winskel. Models and Logic of MOS Circuits. In Manfred Broy, editor, *Logic of Programs and Calculi of Discrete Design*, volume 36 of *NATO Advanced Studies Institute Series F*. Springer-Verlag, 1987. Proceedings of the NATO Advanced Study Institute on Logic of Programs and Calculi of Discrete Design held in Marktoberdof, FDR, July 29–August 10, 1986.

[137] Glynn Winskel. A compositional model of MOS transistors. In Birtwistle and Subrahmanyam [13].

# Appendix A — BNF rules for ASDL

The context-free syntax of ASDL definitions is as follows (delimiters are denoted by words in bold and terminals by capitalized words):

```
%right-assoc  :< ++
%left-assoc   + -  /* less priority */
%left-assoc   / * :>

module : defs
       ;
defs : defs def
     | /* empty */
     ;

def : VAR formal <= rhs
    ;
formal : formal pat
       | /* empty */
       ;
pat : pat :< pat  /* cons patterns */
    | pat :> pat  /* Snoc patterns */
    | apat
    ;
apat : ( patsl )
     | [ pats ]
     | NUM  /* integers */
     | LV   /* logic values:  H,L,Z,X  */
     | VAR  /* variables with lexical syntax: [a-z][a-zA-Z0-9_]*  */
     | _    /* WILDACARD */
     ;
pats :  /* empty */
     | patsl
     ;
patsl : pat , patsl
      | pat
      ;
rhs : exp terminator
    | exp wheredefs terminator
    ;

terminator : OFFSIDE
           ;
/* infix operators */
exp : exp :< exp
```

```
    | exp :> exp
    | exp ++ exp
    | exp + exp
    | exp - exp
    | exp * exp
    | exp / exp
    | aexp  /* atomic expressions */
    ;
aexp : simple
     | aexp simple /* application is left-associative */
     ;
simple : rec VAR
       | ( expsl )
       | [ exps ]
       | VAR
       | NUM
       | LV
       ;
exps : /* empty */
     | expsl
     ;
expsl : exp , exps
      | exp
      ;
wheredefs : where localdefs
          ;

localdefs : localdef
          | localdefs localdef
          ;

localdef : ipat <= rhs
         ;
ipats : ipat , ipats
      | ipat
      ;
ipat : ( ipats )
     | VAR
     ;
```

The syntax of ASDL atom and bus resolution function specifications used in library interfaces is (using the same conventions as before):

```
%non-assoc  ,
%right-assoc  ->

sig   : sig tspec
      | /* empty */
      ;
tspec   : VAR :: type OFFSIDE   /* atoms and join functions */
        | VAR :: type OFFSIDE   /* basetypes */
        ;
type   : argtype
       | type -> type
       ;

argtype : typename
        | TVAR  /* type variable */
        | ( type_list )
        | [ type ]
        ;
type_list : type , type_list
          | type
          ;
typename : VAR
         ;
```

# Appendix B Proof that *list* to *tree* coercions are bijections

Let's recall the definition of `splitAt` from 4.2:

```
splitAt 0       xs     => ([],xs)
splitAt _       []     => ([],[])
splitAt (n+1) (x:<xs) => (x:<xs1,xs2)
                         where

                         (xs1,xs2) <= rec n xs
```

We first prove that *cat2list (list2cat as)* == *as* for all finite and total lists *as*. Unfortunately, total structural induction on the standard prefix (Cons) ordering on the list *as* won't work. Therefore, we define a well-founded ordering relation > such that, for all finite and total, non-empty lists *as,bs*: *as* ++ *bs* > *as* and *as* ++ *bs* > *bs*. The basis for > is *[a]*. We will also make use of the following *Lemma* whose proof is omitted as obvious:

```
for all finite and total as, for all m,

as == xs ++ ys where (xs,ys) <= splitAt m as
```

*Case of (as = [])* is trivial:

```
cat2list (list2cat [])

{list2cat(1)}

= cat2list Nilt

{cat2list(1)}

=[]
```

*Base Case (as=[a])*:

```
cat2list (list2cat [a])

{list2cat(2)}
```

```
=cat2list (Unit a)
{cat2list(2)}
=[a]
```

The *Inductive Assumption* is that:

```
cat2list (list2cat as) = as for all finite, total and non-empty as
```

*(A1)*

```
cat2list (list2cat bs) = bs for all finite, total and non-empty bs
```

*(A2)*

We have to show that:

```
cat2list (list2cat (as ++ bs)) = as ++ bs
```

The LHS of this equation is unfolded:

```
cat2list ((list2cat cs) Cat (list2cat ds))
where
(cs,ds) <= splitAt m (as++bs)
m = length (as++bs) div 2
{cat2list(3)}
=cat2list (list2cat cs) ++ cat2list (list2cat ds)
{A1, A2 are valid for all as, bs}
= cs ++ ds
{Lemma}
= as ++ bs
```

The second part of the proof, namely that for all binary trees *x*, *list2cat (cat2list x)* == *x*, is a trivial application of total structural induction on binary trees.

# Appendix C — Proof of theorem 5.3.1

In the following, the proof of Theorem 5.3.1 and of the `scanl` to `scanlt` transformation are given. The first step, which is probably the most demanding, is to extend 'scan' into an attribute grammar, a task that is always feasible according to proposition 5.2, although the degree of success depends critically on the particular selection of the inherited attributes. Therefore, we define:

```
scanlt1 @ e t r => afst (map (@ r)) (scanl @ e t)  (1)
```

We calculate this definition by cases:

```
scanlt1 @ e Nilt r
{unfold (1)}
= afst (map (@ r)) (Nilt,e)
{definitions of afst, map}
= (Nilt,e)
scanlt1 @ e (Unit a) r
{unfold (1)}
= afst (map (@ r)) (Unit e,a)
{definitions of afst, map}
= (Unit (e @ r),a)
{e is left-unit of @}
= (Unit r,a)
scanlt1 @ e (x Cat y) r
{unfold (1)}
```

```
= afst (map (@ r)) (xt Cat yt, xv @ yv)
  where
   (xt,xv) <= scanl @ e x
   (yt,yv) <= afst (map (@ xv)) (scanl @ e y)
{definitions of afst, map}
= ((map (@ r) xt) Cat (map (@ r) xt), xv @ yv)
  where
   (xt,xv) <= scanl @ e x
   (yt,yv) <= afst (map (@ xv)) (scanl @ e y)
{definition of afst}
= (xt1 Cat yt1, xv @ yv)
  where
   (xt1,xv) <= afst (map (@ r)) (scanl @ e x)
   (yt1,yv) <= afst (map (@ r) . map (@ xv)) (scanl @ e y)
{map distribution over composition}
= (xt1 Cat yt1, xv @ yv)
  where
   (xt1,xv) <= afst (map (@ r)) (scanl e x)
   (yt1,yv) <= afst (map ((@ r) . (@ xv))) (scanl @ e y)
{@ is left associative}
= (xt1 Cat yt1, xv @ yv)
  where
   (xt1,xv) <= afst (map (@ r)) (scanl e x)
   (yt1,yv) <= afst (map (@ (r @ xv))) (scanl @ e y)
{fold using (1)}
```

```
= (xtl Cat ytl, xv @ yv)
  where
  (xtl,xv) <= scanltl @ e x r
  (ytl,yv) <= scanltl @ e y (r @ xv)
```

Assembling all these results and folding using the semantics of the rec construct, we get the definition:

```
scanltl @ e Nilt r => (Nilt,e)
scanltl @ e (Unit a) r => (Unit r,a)
scanltl @ e (x Cat y) r => (xt Cat yt, xv @ yv)
                           where
                           (xt,xv) <= rec x r
                           (yt,yv) <= rec y (r @ xv)
```

**Corollary:** The validity of the transformation of `scanl` into `scanlt` is justified as follows:

```
scanl @ e t
{map (@ e) is the identity on finite lists parameterized by the type
of e}
= afst (map (@ e)) (scanl @ e t)
{definition of scanltl}
= scanltl @ e t e
{definition of scanlt}
= scanlt @ e t
```

End of Corollary

In the following calculation, we make use of the law (*map-scanlt1*):

```
(map k || k) . (scanlt1 @ e t r) == scanlt1 $ e t (k r)
when k (a @ b) == (k a) $ b
```

This is a straightforward application of the *map-scanl* lemma whose dual *(map-scanr)* was proved in section 4.2.3.

With a little hindsight, we define:

```
iscanl k @ $ e t r => afst (map k) (scanlt1 @ e t r)  (2)
```

We then "decouple" the synthesized attributes of the right-hand-side of *(2)*[1]:

```
iscanl k @ $ e t r
{unfold using (2), afst}
= (fst ((map k || k) (scanlt1 @ e t r)), snd (scanlt1 @ e t r))
{map-scanlt1 law}
= (fst (scanlt1 $ e t (k r)), snd (scanlt1 @ t r))  (3)
Calculating now 'iscanl' by cases, we have:
iscanl k @ $ e Nilt r
{unfold using (2),...}
= (Nilt,e)
iscanl k @ $ e (Unit a) r
{unfold using (2),...}
= (Unit (k r),a)
iscanl k @ $ e (x Cat y) r
{unfold using (2), and the definitions of map,afst}
= (xt Cat yt, xv @ yv)
  where
```

[1] Assuming the projectors: `fst (a,b) => a,  snd (a,b) => b.`

```
  (xt,xv) <= afst (map k) (scanlt1 @ e x r)
  (yt,yv) <= afst (map k) (scanlt1 @ e y (r @ xv))
{map-scanlt1 law}
= (xt Cat yt, xv @ yv)
  where
  (xt,xv) <= (fst (scanlt1 $ e x (k r)), snd (scanlt1 @ e x r))
  (yt,yv) <= (fst (scanlt1 $ e x (k (r @ xv))), snd (scanlt1 @ e x r))
= (xt Cat yt, xv @ yv)
  where
  (xt,xv) <= (fst (scanlt1 $ e x (k r)), snd (scanlt1 @ e x r))
  (yt,yv) <= (fst (scanlt1 $ e x ((k r) $ xv)), snd (scanlt1 @ e x r))
{fold using (3)}
= (xt Cat yv,xv @ yv)
  where
  (xt,xv) <= iscanl k @ $ e x (k r)
  (yt,yv) <= iscanl k @ $ e y ((k r) $ xv)
```

By assembling the above results and employing the semantics of rec, we obtain:

```
iscanl k @ $ e Nilt r => (Nilt, e)
iscanl k @ $ e (Unit a) r => (Unit (k r),a)
iscanl k @ $ e (x Cat y) r=> (xt Cat yt, xv @ yv)
```

```
    where
    (xt,xv) <= rec x (k r)
    (yt,yv) <= rec y ((k r) $ xv)
```

Finally, the proof of the theorem is established by simple case analysis that the left-hand-side of the theorem, `iscanl k @ $ e t e`, is equivalent to the right-hand-side, without even needing induction. We will omit this trivial step.

# Appendix D — Proof of proposition *ntranI*

We re-state the proposition to be proved:

**Proposition** *(ntranI)* Assuming that all inputs as are driven, the following equivalences are tautologies in the instantaneous behavioral model:

1. `Ntran (And as,x) == n_series (as,x)` *(ntranIseries)*
2. `Ntran (Or as,x) == n_para (as,x)` *(ntranIpara)*

where *length as*≥2.

The proof is by induction on the list as. According to definition *Eq MOSterm*, we have to prove that: *(I)* the validity conditions are equivalent, *(II)* the drive conditions are equivalent, and *(III)* both the Left-Hand-Side (from now on, the LHS) and the Right-Hand-Side (from now on, the RHS) evaluate to the same values in the instantaneous behavioral model in environments that satisfy the same drive conditions. In the following discussion, we will refer to the condition that all as are driven as the "drive assumption".

We proceed by proving *ntranIseries* first.

*Base Case:*

```
Ntran (And [a1,a2],x) == n_series ([a1,a2],x)
{n_series}
= Ntran (And [a1,a2],x) == Ntran (a1, Ntran (a2,x)
```

For the validity condition of the LHS, we have:

```
valid' (Ntran (And [a1,a2],x)) e
{valid' Ntran}
= valid' x e
```

For the RHS, we have similarly:

```
valid' (Ntran (a1, Ntran (a2,x))) e
{valid' Ntran, ...}
= valid' x e
```

Hence, part *(I)* of the proof has been completed.

For the drive condition of the LHS, we have:

```
driven' (Ntran (And [a1,a2],x)) e
{driven' Ntran}
= driven' (And [a1,a2]) e && eval' (And [a1,a2]) e == H &&
  driven' x e && eval' x e == L
{driven' And, drive assumption}
= eval' (And [a1,a2]) e == H && driven' x e && eval' x e == L
```

Similarly, for the RHS:

```
driven' (Ntran (a1, Ntran (a2,x))) e
{driven' Ntran, ...}
= driven' a1 e && eval' a1 e == H &&
  driven' (Ntran (a2,x)) e && eval' (Ntran (a2,x)) e == L
{driven' Ntran}
= driven' a1 e && eval' a1 == H && eval' (Ntran (a2,x)) e == L &&
  driven' a2 e && eval' a2 == H && driven' x e && eval' x e == L
{eval' Ntran, drive assumption, ...}
= eval' a1 == H && eval' a2 == H && driven' x e && eval' x e == L
{eval' And, ...}
= eval' (And [a1,a2]) e == H && driven' x e && eval' x e == L
```

Call this formula *ntranIseriesV*. Part *(II)* of the base case proof has been demonstrated.

The verification that, `eval' (Ntran (And [a1,a2],x)) e == eval' (n_series ([a1,a2],x)) e` when *ntranIseriesV* is trivially performed using case analysis on MOS values. This completes part *(III)* of the proof for the base case.

*Inductive Step:* The generalization of the above proof to an arbitrary number of inputs is (trivially) done using the fact that And is associative. Q.E.D.

We now proceed by proving *ntranIpara*

*Base case:*

```
Ntran (Or [a1,a2],x) == n_para ([a1,a2],x)
{unfold n_para}
= Ntran (Or [a1,a2],x) e == b1 Join b2
                              where
                              b1 <= Ntran (a1,x)
                              b2 <= Ntran (a2,x)
```

The validity condition of the LHS is:

```
valid' (Ntran (Or [a1,a2],x) e
{valid' Ntran, driven' a1,a2}
= valid' x e
```

Similarly, the validity condition of the RHS is:

```
valid' (b1 Join b2
        where
        b1 <= Ntran (a1,x)
        b2 <= Ntran (a2,x) ) e
{introduce abbreviations (1)}
= valid' (b1 Join b2) e'
```

```
  where (1)
  e' = e ++ [(b1,(eb1,db1)),(b2,(eb2,db2))]
  eb1 = eval' (Ntran (a1,x)) e'
  db1 = driven' (Ntran (a1,x)) e'
  eb2 = eval' (Ntran (a2,x)) e'
  db2 = driven' (Ntran (a2,x)) e'
{valid' Join}
= valid' b1 e' && valid' b2 e' &&
  not (eb1==H && eb2==L) && not (eb1==L && eb2==H)
{case analysis on eval', (1), introduce abbreviations (2)}
= not (ea1/=L && ex==H && ea2/=L && ex==L) && not (eb1==L && eb2==H)
  where (2)
  ea1 = eval' a1 e'
  ea2 = eval' a2 e'
  ex = eval' x e
{Eq MOSval, ...}
= valid' b1 e' && valid' b2 e' &&
  not (False && ea1/=L && ea2/=L) && not (eb1==L && eb2==H)
{simplification, repeat the same on remaining not clause}
= valid' b1 e' && valid' b2 e'
{valid' Ntran, (1), (2), ...}
= valid' x e'
```

The drive condition, now, of the LHS is:

```
driven' (Ntran (Or [a1,a2],x)) e
{driven' Ntran}
```

```
= driven' (Or [a1,a2]) e && eval' (Or [a1,a2]) e == H && driven' x e
  && eval' x e == L
{driven' Or, drive assumption}
= eval' (Or [a1,a2]) e == H && driven' x e && eval' x e == L
```

Similarly for the RHS:

```
driven' (Local ([b1,b2],[Ntran (a1,x),Ntran (a2,x)]) (b1 Join b2)) e
{driven' Local, use (1) above}
= driven' (b1 Join b2) e'
  where (1)
{driven' Join}
= (driven' b1 e' || driven' b2 e') && valid' (b1 Join b2) e'
{valid' (b1 Join b2), use (2) above}
= (driven' a1 e' && ea1==H && driven' x e' && ex==L ||
   driven' a2 e' && ea2==H && driven' x e' && ex==L) && valid' x e'
{eval' Or, drive assumption, ...}
= driven' x e && eval' x e == L && eval' (Or [a1,a2]) e == H
```

Call this formula *ntranIparaV*. Parts *(I)* and *(II)* of the proof of the base case have now been completed.

Part *(III)*, i.e. `eval' (Ntran (Or [a1,a2],x)) e ==` `eval' (n_para ([a1,a2],x)) e` when *ntranIparaV* `== True`, is easily proved by case analysis. This completes the proof for the base case.

*Inductive Step:* The inductive step is easily shown using the fact that `Or` is associative. This completes the proof that *ntranIpara* holds for any number of inputs as. Q.E.D.

# Appendix E — Proof of proposition *ntranR*

We re-state the proposition to be proved:

**Proposition** *(ntranR)* Assuming that fN is the equivalent boolean formula of a N-block with inputs as, then

```
(Ntran (fN as,Ntran (a,Gnd))) Join x ==
(Ntran (fN as,Not a)) Join x
```

when not (eval' (fN as) e == H && eval' a e == L &&

```
      eval' x e == L)
```

We will employ *Eq MOSterm* in order to prove the above proposition. Define:

```
ntranRvalid = not (eval' (fN as) e == H && eval' a e == L
              && eval' x e == L)
```

The validity condition of the LHS is:

```
valid' ((Ntran (fN as,Ntran (a,Gnd))) Join x) e

{introduce abbreviations (1)}

= valid' (w1 Join x) e'

  where (1)

  e' = [(w,(ew,dw)),(w1,(ew1,dw1))] ++ e

  ew = eval' (Ntran (a,Gnd)) e'

  dw = driven' (Ntran (a,Gnd)) e'

  ew1 = eval' (Ntran (fN as,w)) e'

  dw1 = driven' (Ntran (fN as,w)) e'

  fNon = eval (fN as) e' == H

  ex = eval' x e'
```

```
{valid' Join}
= valid' wl e' && valid' x e' &&
  not (ewl==H && ex==L) && not (ewl==L && ex==H)
{case analysis on eval' fN-block, (1)}
= valid' wl e' && valid' x e' &&
  not (fNon e' && ew==H && ex==L) && not (ewl==L && ex==H)
{case analysis on eval' Ntran, ...}
= valid' wl e' && valid' x e' &&
  not (fNon e' && False && ex==L) && not (ewl==L && ex==H)
{case analysis on eval' fN-block, ...}
= valid' wl e' && valid' x e' && not (fNon e' && ew==L && ex==H)
{case analysis on eval' Ntran, assumption that 'a' is driven, ...}
= valid' wl e' && valid' x e' &&
  not (fNon e' && eval' a e' == H && eval' x e == H)
{valid' wl e' == valid' w e' == True, (1), ...}
= valid' x e && not (eval' (fN as) e == H && eval' a e == H && eval' x e
== H)
```

The drive condition of the LHS is:

```
driven' ((Ntran (fN as,Ntran (a,Gnd))) Join x) e
{driven' Join, (1) above}
= (driven' wl e' || driven' x e') && valid' (wl Join x) e'
{derivation of valid' LHS}
= (driven' wl e' || driven' x e') && valid' x e' &&
  not (fNon e' && eval' a e' == H && eval' x e' == H)
{driven' fN-block, (1), ...}
```

```
= (fNon e' && eval' w e' == L && driven' w e' || driven' x e') &&
  valid' x e' && not (fNon e' && eval' a e' == H && eval' x e == H)
{eval' Ntran, assumption that 'a' is driven, ...}
= (fNon e' && eval' a e' == H || driven' x e') &&
  valid' x e' && not (fNon e' && eval' a e' == H && eval' x e == H)
{(1), ...}
= (driven' x e && not (eval' (fN as) e == H && eval' a e == H && eval' x e
== H)) ||
(valid' x e && eval' (fN as) e == H && eval' a e == H && eval' x e /= H)
```

The validity condition of the RHS, now, is:

```
valid' ((Ntran (fN as,Not a)) Join x) e
{introduce abbreviations (2)}
= valid' (w2 Join x) e'
  where (2)
    e' = [(w2,(ew2,dw2))] ++ e
  ew2 = eval' (Ntran (fN as,Not a)) e'
  dw2 = driven' (Ntran (fN as,Not a)) e'
  fNon = eval' (fN as) e' == H
  ex = eval' x e'
{valid' Join, (2)}
= valid' x e' && valid' w2 e' &&
  not (ew2==H && ex==L) && not (ew2==L && ex==H)
{eval' fN-block twice, ...}
= valid' x e' && valid' w2 e' &&
  not (eval' (Not a) e' == H && fNon e' && ex==L) &&
```

```
  not (eval' (Not a) e' == L && fNon e' && ex==H)
{eval' Not, valid' Not, assumption that 'a' is driven, (1), ...}
= valid' x e &&
  not (eval' a e == L && eval' (fN as) e == H && eval' x e == L) &&
  not (eval' a e == H && eval' (fN as) e == H && eval' x e == H)
{ntranRvalid == True assumption}
= valid' x e && not (eval' a e == H && eval' (fN as) e == H && eval' x e
== H)
```

The drive condition of the RHS is:

```
driven' ((Ntran (fN as,Not a)) Join x) e
{driven' Local, use (2)}
= driven' (w2 Join x) e'
  where (2)
{driven' Join}
= (driven' w2 e' || driven' x e') && valid (w2 Join x) e'
{derivation of valid' RHS (above)}
= (driven' w2 e' || driven' x e') &&
  valid' x e && not (eval' a e == H && eval' (fN as) e == H && eval' x e
== H)
{driven' Ntran, eval' Ntran-block, eval' Not, ...}
= (fNon e' && eval' a e' == H || driven' x e') &&
  valid' x e && not (eval' a e == H && eval' (fN as) e == H && eval' x e
== H)
= (driven' x e && not (eval' (fN as) e == H && eval' a e == H && eval' x
e == H))
```

```
= (valid' x e && eval' (fN as) e == H && eval' a e == H && eval' x e /= H)
```

By examining the above equations, we see that parts *(I)* and *(II)* of *Eq MOSterm* have been completed. The remaining part, i.e. that both sides evaluate to the same values when the drive condition holds, is trivially proved by case analysis. Q.E.D.

# Appendix F — Proof of proposition *net_create*

We re-state the proposition to be proved:

**Proposition** *(net_create)* When x is always driven, then:`f x ==`

`(Ptran (Not (f x),Pwr)) Join (Ntran (Not (f x),Gnd))`

where `f` is a completely specified boolean formula.

In the following, we make use of definition *Eq MOSterm*. First, we examine the validity condition of the RHS.

```
valid' ((Ptran (Not (f x),Pwr)) Join (Ntran (Not (f x),Gnd))) e
{introduce abbreviations (1)}
= valid' (a Join b) e'
  where (1)
  a = Ptran (Not (f x),Pwr)
  b = Ntran (Not (f x),Gnd)
  e' = e ++ [(a,(ea,da)),(b,(eb,db))]
  ea = eval' a e'
  da = driven' a e'
  eb = eval' b e'
  db = driven' b e'
{valid' Join}
= valid' a e' && valid' b e' &&
  not (ea==H && eb==L) && not (ea==L && eb==H)
{(1), eval' Ntran, eval' Gnd, eval' Ptran, eval' Pwr}
= valid' a e' && valid' b e' &&
```

```
  not (eval' (Not (f x)) e' /= L && True &&
  eval' (Not (f x)) e' /= H && True) && not False
{eval' Not, De Morgan, ...}
= valid' a e' && valid' b e' &&
   (eval' (f x) e' == L || eval' (f x) e' == H)
{f is total, x is driven}
= valid' a e' && valid' b e'
{(1), valid' Ptran, valid' Ntran}
= valid' Pwr e' || valid' Gnd e'
{...}
= True
```

Next, we examine the drive condition of the RHS.

```
driven' ((Ptran (Not (f x),Pwr)) Join (Ntran (Not (f x),Gnd))) e
{(1) above, ...}
= driven' (a Join b) e'
{driven' Join}
= (driven a e' || driven b e') && valid' (a Join b) e'
{derivation of valid' RHS above, ...}
= (driven a e' || driven b e')
{driven' Ntran, driven' Ptran, eval' Pwr, eval' Gnd, (1), ...}
= (driven' (Not (f x)) e' && eval' (Not (f x)) e' == L && True) ||
   (driven' (Not (f x)) e' && eval' (Not (f x)) e' == H && True)
{driven' Not, assumption, eval' Not, ...}
= eval' (f x) e' == H || eval' (f x) e' == L
{f is completely specified}
```

```
= True
```

Since f is a completely specified boolean function, the drive and validity condition of the LHS are trivially satisfied. Hence, parts *(I)* and *(II)* of the *Eq MOSterm* definition have been completed. Finally, we examine part *(III)* of the Eq MOSterm definition. The value of the RHS is:

```
eval' ((Ptran (Not (f x),Pwr)) Join (Ntran (Not (f x),Gnd))) e
{factorize (Not (f x)), Not == inv}
= eval' (f x) e
```

which is the value of the LHS as well.

# Appendix G The CMOS rewriting package in Haskell

The following module is the core of the CMOS synthesizer:

```
module Cond_rewrite where
data Mosterm = Var Varnum
             | Pwr
             | Gnd
             | Ntran (Mosterm, Mosterm)
             | Ptran (Mosterm, Mosterm)
             | Join Mosterm Mosterm
             | And [Mosterm]
             | Or [Mosterm]
             | Not Mosterm
             | Local [(Mosterm,Mosterm)] Mosterm
             | Zz -- pseudo-term (used for initialization)

type Varnum = Int

instance Eq Mosterm where
   Var a == Var b              = a==b
   Not (Var a) == Not (Var b) = a==b
   Join a b == Join a' b'      = a==a' && b==b'
   a == b | otherwise          = False

-------------------------------------------------------------------------

-- Create a CMOS NW from the Specification fN=(Output==L) & fP=(Output==H)
-- (DON'T CARES when neither fN or fP is set)
net_create (fN,fP) = flat (net_P `Join` net_N)
                     where
                     net_N = reN (max_net fN) (Ntran (fN,Gnd))
                     net_P = reP (max_net net_N) (Ptran (fP',Pwr))
                     -- TEMP FIX -- use inherited attrs instead
                     fP' = deM (Not fP)

-- De Morgan law
deM (Not (And xs)) = Or (map deM (map Not xs))
deM (Not (Or xs))  = And (map deM (map Not xs))
deM (Not (Not x))  = deM x
deM a  | otherwise = a

notNot (Not (Not a))  = a
notNot x | otherwise  = x


-------------------------------------------------------------------------
------------------ Unconditional Transformations ------------------------
-------------------------------------------------------------------------

-- N-part of the NW
-- There should be some restriction to the number of transistors in series,
-- e.g. less than 6, which could be imposed as an integrity constraint
reN n (Ntran (And [a],x))
 = reN (max (max_net a') n) (Ntran (a',x))
   where
   a' = reN n a
```

```
reN n (Ntran (And (a:as),x))
 = reN (maximum [n,max_net x',max_net a']) (Ntran (a',x'))
   where
   a' = reN n a
   x' = reN (max (max_net a') n) (Ntran (And as,x))
reN n (Ntran (Or [a],Gnd))
 = reN (max n (max_net a')) (Ntran (a',Gnd))
   where
   a' = reN n a
reN n (Ntran (Or (a:as),Gnd))
 = left `Join` (reN (maximum [max_net left,n,max_net a']) (Ntran (Or as,Gnd)))
   where
   a' = reN n a
   left = reN (max (max_net a') n) (Ntran (a',Gnd))
-- FANOUT! inherited attributes should be used instead of max_net
-- inputs as assumed to be Literals
reN n (Ntran (Or as,x))
 = Local [(Var (n+1),xdeep)] (reN (max (max_net xdeep) (n+1)) (foldl1 Join ys))
   where
   xdeep = reN (n+1) x
   ys = map ntran as
   ntran a = Ntran (a,x')
   x' = Var (n+1)
reN _ other  = other

-- P-part of the NW
reP n (Ptran (Or [a],x))
 = reP (max (max_net a') n) (Ptran (a',x))
   where
   a' = reP n a
reP n (Ptran (Or (a:as),x))
 = reP (maximum [max_net x',n,max_net a']) (Ptran (a',x'))
   where
   a' = reP n a
   x' = reP (max (max_net a') n) (Ptran (Or as,x))
reP n (Ptran (And [a],Pwr))
 = reP (max (max_net a') n) (Ptran (a',Pwr))
   where
   a' = reP n a
reP n (Ptran (And (a:as),Pwr))
 = left `Join` (reP (maximum [max_net left,n,max_net a']) (Ptran (And as,Pwr)))
   where
   a' = reP n a
   left = reP (max (max_net a') n) (Ptran (a',Pwr))
 = Local [(Var (n+1),xdeep)] (reP (max (max_net xdeep) (n+1)) (foldl1 (Join) ys))
   where
   xdeep = reP (n+1) x
   ys = map ptran as
   ptran a = Ptran (a,x')
   x' = Var (n+1)
reP _ other    = other

-------------------------------------------------------------------------
---------- Conditional Transformations ----------------------------------
-------------------------------------------------------------------------

-- We don't rewrite gates because these are connected to the primary inputs.
-- No new nets introduced in this step, only eliminated.
-- Again, the assumption is that the RHS has been flattened.
-- Commutativity of And, Or is not accounted for!

-- Note new environment
re e cond (x `Join` y)
 = (re (e `joine` y) cond x) `Join` (re (e `joine` x) cond y)
```

```
re e cond (Ntran (b, Ntran (a,Gnd)))
 | forall (free_vars e) cond [notNot (Not a),b] e H
 = Ntran (b, notNot (Not a))
re e cond (Ntran (c,Ntran (b,Ntran(a,Gnd))))
 | forall (free_vars e) cond [notNot (Not a),b,c] e H
 = Ntran (c, Ntran (b, notNot (Not a)))
re e cond (Ptran (b, Ptran (a,Pwr)))
 | forall (free_vars e) cond [a,notNot (Not b)] e L
 = Ptran (b, notNot (Not a))
re e cond (Ptran (c,Ptran (b,Ptran(a,Pwr))))
 | forall (free_vars e) cond [a,notNot (Not b),notNot (Not c)] e L
 = Ptran (c, Ptran (b, notNot (Not a)))
re e cond x | otherwise = x

-- Finally!
re_cond cond (Local ds top)
 = Local ds (re cond (Local ds Zz) top) -- rewrite top expr only
re_cond cond expr
 = re Zz cond expr -- the initial environment is providing Zz

-- The top level invocation
synth cond = re_cond cond . net_create

-------------------------------------------------------------------------
----------------- Abstract Interpretation -------------------------------
-------------------------------------------------------------------------

data State = X | H | L | Z

instance Ord State where
     Z <=   = True
     H <= X = True
     L <= X = True
     x <= y | x==y      = True
            | otherwise = False

instance Eq State where
      X == X = True
      H == H = True
      L == L = True
      Z == Z = True
      x == y | otherwise = False

-- Evaluate a MOS term in a given environment
-- All MOS terms have a single output.
eval:: Mosterm -> [(Varnum,State)] -> State
eval Pwr e   = H
eval Gnd e   = L
eval Zz  e   = Z -- used for initial environments
eval (Ntran (g,s)) e
  | gate == H = eval s e
  | gate == L = Z
  | otherwise = X
    where
    gate = eval g e
eval (Ptran (g,s)) e
  | gate == L = eval s e
  | gate == H = Z
  | otherwise = X
    where
    gate = eval g e
eval (a `Join` b) e -- least upper bound of a,b
  | ea == eb  = ea
  | eb == Z   = ea
  | ea == Z   = eb
```

```
  | otherwise = X
    where
    ea = eval a e
    eb = eval b e
eval (Var n) e = lu n e
eval (Not x) e
 = no (eval x e)
    where
    no H = L
    no L = H
    no X = X
    no Z = error ("undriven input at Not ")
eval (Local ds t) e
 = eval t e'
    where
    e' = e++map (\(Var n,r)->(n,eval r e')) ds
-- the following are used for validation of input conditions only
eval (And xs) e
  | any (==X) (map (\x -> eval x e) xs) = X
  | any (==Z) (map (\x -> eval x e) xs) = X
  | all (==H) (map (\x -> eval x e) xs) = H
  | otherwise  = L  -- at least one is  L
eval (Or xs) e
  | any (==X) (map (\x -> eval x e) xs) = X
  | any (==Z) (map (\x -> eval x e) xs) = X
  | any (==H) (map (\x -> eval x e) xs) = H
  | otherwise  = L -- all xs are L

-- lookup a value in the environment
lu:: Varnum -> [(Varnum,State)] -> State
lu n ((n',v):e)
  | n==n'     = v
  | otherwise = lu n e

-- repetitions allowed (not used for Local defs)
vars_in (Var a)       = [a]
vars_in (Not (Var a)) = [a] -- only gates can be connected to negated signals
vars_in (a `Join` b)  = vars_in a ++ vars_in b
vars_in (Ntran (g,s)) = vars_in g ++ vars_in s
vars_in (Ptran (g,s)) = vars_in g ++ vars_in s
vars_in (And xs)      = concat (map vars_in xs)
vars_in (Or xs)       = concat (map vars_in xs)
vars_in _             = [] -- Pwr or Gnd

-- Assume one level deep local scope only! nub == mkset
free_vars (Local ds t)
 = nub ((avs ++ vars_in t) \\ bvars)
    where
    bvars = map (\(Var n,_)->n) ds -- bound vars in the local scope
    avs = concat (map (\(Var n,rhs)->vars_in rhs) ds)
free_vars e = nub (vars_in e)

------------------------ Auxiliary Functions --------------------------

-- Array_left
accl:: (a->b->a, a->b->c) -> a -> [b] -> ([c],a)
accl (opx,opy) ix [] = ([],ix)
accl (opx,opy) ix xs = (ys++[y],w)
                       where
                       (x,a) = (init xs,last xs)
                       y = ix' `opy` a
                       w = ix' `opx` a
                       (ys,ix') = accl (opx,opy) ix x
```

```
-- Generate all possible environment bindings for the free variables
-- (permutations of H,L)
gen_env ns = transp (fst (accl (opx,opy) 0 ns))
             where
             k = 2^(length ns)
             opx m n = m+1
             opy m n = map (pair n) (take k (clk m))
             pair n v = (n,v)

clk n = copy (2^n) H ++ copy(2^n) L ++ clk n

distl a []     = []
distl a (x:xs) = (a++x):(distl a xs)

-- list-of-lists transposition
transp x | x'==[]    = []
         | otherwise = map head x':transp (map tail x')
           where
           x' = takeWhile (/=[]) x

forall:: [Varnum] -> Mosterm -> [Mosterm]->Mosterm->State->Bool
forall fvs cond constraints expr val
 = and (map g env)
   where
   g e = (eval cond e==L)||(eval expr e==val) -- cond `implies` expr==val
   freeVars = fvs \\ (concat (map vars_in constraints))
   es = gen_env freeVars
   ec = concat (map f constraints)
   f (Var x)       = [(x,H)]
   f (Not (Var x)) = [(x,L)]
   f _             = [] -- Gnd or Pwr
   env = distl ec es -- attach constraints to each free var environment


-- this is used in re
joine (Local ds t) e = Local ds (t `Join` e)
joine e (Local ds t) = Local ds (t `Join` e)
joine (Local ds t) (Local ds' t') = Local (ds++ds') (t `Join` t')
joine e1 e           = e1 `Join` e

max_net (Var n)  = n
max_net (Not a)  = max_net a
max_net (And xs) = maximum (map max_net xs)
max_net (Or xs)  = maximum (map max_net xs)
max_net (a `Join` b)   = max (max_net a) (max_net b)
max_net (Ntran (g,s))  = max (max_net g) (max_net s)
max_net (Ptran (g,s))  = max (max_net g) (max_net s)
max_net (Local ds top) = max (max_net top) (maximum (map max_net (map snd ds)))
max_net _ = 0


-- "lift" local defs to the top level
flat a = case flatten (a,[]) of
           (top,[]) -> a
           (top,ds) -> Local ds top

flatten (a `Join` b, ds)
 = (a' `Join` b', ds ++ da ++ db)
   where
   (a', da) = flatten (a,[])
   (b', db) = flatten (b,[])
-- There is only one local definition per expression after reP and reN
flatten (Local [(n1,Local [(n2,d2)] t2)] t1,ds)
 = flatten (Local [(n2,d2)] t1, (n1,t2):ds)
flatten (Local ds' t, ds)
```

```
 = (t,ds'++ds)
flatten (Ntran (g,s),ds)
 = (Ntran (g,s'), ds' ++ ds)
   where
   (s',ds') = flatten (s,[])
flatten (Ptran (g,s),ds)
 = (Ptran (g,s'), ds' ++ ds)
   where
   (s',ds') = flatten (s,[])
flatten (a,ds)  = (a,ds)
```

# Appendix H — A modular translator from the wirelist intermediate form to VHDL

```
A wirelist to VHDL translator in Miranda literate mode:

>%export layout def comp port net
>%free { module_name :: [char];}

The syntax of parsed type (interface) delarations:

>netlist == [def] || anticipate future types of definitions
>def ::= Def comp_name [generic] [port] [comp_instance]
>generic == (param_name,type_name)
>param_name == num
>comp_instance == (instance_num, comp)

Library or user defined component are NOT assumed unidirectional in
anticipation of future versions of ASDL. In this version, user defined
components are not allowed to have attributes.

>comp ::= Lib comp_name [attr_value] [net]
>       | Fun comp_name [attr_value] [net]
>       | Constant attr_value type_name [net]
>attr_value == string
>port ::= In port_name type_name | Out port_name type_name
>       | InOut port_name type_name || not in use in this version
>port_name == num
>net ::= LVAR num | OPEN | PWR | GND

Convenient type synonyms

>string == [char]
>comp_name == string
>type_name == string
>instance_num == num

Global Variables.

Beware, the next function uses side-effects (on purpose)!
'fst_of_three' forces evaluation of system call first because it
examines the exit status. ATTENTION! Do not modify <basis.m> between the
execution of 'sdl' and 'sdl2vhdl'.

>library:: netlist
>library = seq (force (fst_of_three
>              (system ("libcr "++bdir++"basis.m > "++bdir++"basis.if"))))
>              (readvals (bdir++"basis.if"))
>          where
>          bdir = getenv "SDLDIR" ++ "/struct/"

Read and parse an '*.if' file:
```

```
>module:: netlist
>module = readvals module_name
```

*Conversion to VHDL. Top level function invocation:*

```
>layout
> = "-- This file is automatically generated by running \"sdl2vhdl\"\n-- "
>   ++ "User: " ++ getenv "USER" ++ "\t" ++ fst_of_three (system "date")
>   ++ "\n" ++ mix (map proc_def module) "\n\n" ++ "\n"
```

*Process a single definition from a given "module". Library components are assumed to have VHDL "compatible" names.*

```
>proc_def:: def->[char]
>proc_def (Def name gs ps [])
> = proc_entity (name,gs,ps)
>proc_def (Def name gs ps comps)
> = proc_entity (name,gs,ps) ++ "\n" ++
>   proc_arch (name,ctypes,signals,comps)
>   where
>   (cnames,locals) = proc_comps comps
>   signals = locals -- port_names ps
>   ctypes  = map assoc cnames
```

*Print entity declaration:*

```
>proc_entity:: ([char],[(num,[char])],[port])->[char]
>proc_entity (name,gs,ps)
> = error (beep++"component \""++name++"\" is a VHDL keyword...\n"++
>   "Edit *.sdl file and try again!\n"), if member reserved_vhdl_words name
> = "\nentity "++name++" is "++lay_generics gs ++
>   "\n\t"++lay_ports ps++
>   "\nend "++name++";\n", otherwise
```

*Note: the undersore char '_' is used in generating names for generics, ports and internal nets. Avoid using it in ASDL descriptions!*

```
>lay_generics:: [(num,[char])]->[char]
>lay_generics [] = ""
>lay_generics gs = "generic ("++mix (map show_generic gs) "; "++");"
>                 where
>                 show_generic (n,ty) = "p"++shownum n++": "++ty
```

*Note: it is always true that* `#ports ~= []`

```
>lay_ports:: [port]->[char]
>lay_ports ps = "port (" ++ mix (map show_port ps) "; " ++");"
>              where
>              show_port (In n ty)    = "x"++shownum n++": IN "++ty
>              show_port (Out n ty)   = "x"++shownum n++": OUT "++ty
>              show_port (InOut n ty) = "x"++shownum n++": INOUT "++ty
```

*Print architecture (stuctural) body:*

```
>proc_arch:: ([char],[def],[net],[(num,comp)])->[char]
>proc_arch (name,ctys,signals,cmaps)
> = "architecture struct_" ++ name ++ " of " ++ name ++ " is\n\n" ++
>   lay_ctypes ctys ++ "\n" ++
>   lay_signals signals ++ "\n" ++
```

```
>   "begin\n" ++
>   lay_cmaps cmaps ++ "\n" ++
>   "end struct_" ++ name ++ ";"
```

*Print entities that are used inside annother component:*

```
>lay_ctypes:: [def] -> [char]
>lay_ctypes ctys = "\t" ++ mix (map show_cty ctys) "\n\t"
>                 where
>                 show_cty (Def s gs ps nets)
>                 = "component " ++ s ++ " " ++ lay_generics gs ++ "\n\t"
>                   ++ lay_ports ps ++ "\n\tend component;"
```

*Print local signals -- TEMP FIX because we don't do type checking, assuming all local signals are of type "bit"*

```
>lay_signals:: [net]->[char]
>lay_signals [] = ""
>lay_signals xs = "\n\tsignal "++mix (map show_signal xs) ","++" : bit;\n"
>                 where
>                 show_signal (LVAR n) = "x"++shownum n
>                 show_signal PWR      = "PWR"
>                 show_signal GND      = "GND"
```

*Print instance connections:*

```
>lay_cmaps:: [(num,comp)]->[char]
>lay_cmaps cs = "\n" ++ mix (map show_cmap cs) ";\n" ++";\n"

>show_cmap:: (num,comp)->[char]
>show_cmap (n,Lib s vs ns)
> = ljustify 8 ("\tU"++shownum n++":")++s++" "++gen_map vs++" "++port_map ns
>show_cmap (n,Fun s vs ns)
> = ljustify 8 ("\tU"++shownum n++":")++s++" "++gen_map vs++" "++port_map ns

>gen_map :: [[char]] -> [char]
>gen_map [] = ""
>gen_map vs = "generic map (" ++ mix vs "," ++")"

>port_map:: [net]->[char]
>port_map nets = "port map (" ++ mix (map show_net nets) "," ++ ")"

>show_net (LVAR n) = "x"++shownum n
>show_net OPEN     = "OPEN"
>show_net GND      = "GND"
>show_net PWR      = "PWR"
```

*Utility Functions:*

```
>net_names ((LVAR a):x) = (LVAR a):(net_names x)
>net_names (PWR:x)      = PWR:(net_names x)
>net_names (GND:x)      = GND:(net_names x)
>net_names (OPEN:x)     = net_names x
>net_names []           = []

>port_names ((In a t):x)    = (LVAR a):(port_names x)
>port_names ((Out a t):x)   = (LVAR a):(port_names x)
>port_names ((InOut a t):x) = (LVAR a):(port_names x)
```

```
>port_names []              = []

>proc_comp:: (*,comp)->(comp,[net])
>proc_comp (n,Fun s vs nets) = (Fun s [] [], net_names nets)
>proc_comp (n,Lib s vs nets) = (Lib s [] [], net_names nets)

>proc_comps::[(*,comp)]->([comp],[net])
>proc_comps cs
> = (mkset names, mkset (concat nets)) || Notice mkset
>   where
>   (names,nets) = (map fst ts, map snd ts)
>   ts = map proc_comp cs


>assoc (Fun s [] []) = assoc_def s module
>assoc (Lib s [] []) = assoc_def s library

>assoc_def s ((Def a gs ps cs):x)
>    = Def s gs ps [], if s=a
>    = assoc_def s x, otherwise
>assoc_def other  []
>    = error (beep++"fail to find type of \""
>            ++other ++ "\"... Internal error!\n")

>fst_of_three (stdout,stderr,exit_status)
>    = stdout, if exit_status=0
>    = error (beep++"cannot fork a new process"), if exit_status = -1
>    = error (beep++stderr), otherwise

>mix [] x    = []
>mix [a] x   = a
>mix (a:b) x = a ++ x ++ mix b x

>beep = "\007" || flash the screen and "beep"

Reserved words in VHDL-1076 standard:

>reserved_vhdl_words
> = ["abs","access","after","alias","all","and","architecture","array",
>    "assert","attribute","begin","block","body","buffer","bus","case",
>    "component","configuration","constant","disconnect","downto","else",
>    "elseif","end","entity","exit","file","for","function","generate",
>    "generic","guarded","if","in","inout","is","label","library","linkage",
>    "loop","map","mod","nand","new","next","nor","not","null","of","on",
>    "open","or","others","out","package","port","procedure","process",
>    "range","record","register","rem","report","return","select",
>    "severity","signal","subtype","then","to","transport","type" "units",
>    "until","use","variable","wait","when","while","with","xor"]
```

# Appendix I Wirelist and VHDL description of *scanlt*

The wirelist intermediate form of scanlt16 is:

```
|| This file was automatically generated by executing "sdl"
Def "node" [] [In 1 "bit",In 2 "bit",In 3 "bit",
               Out 4 "bit",Out 1 "bit",Out 5 "bit"]
[(1, Lib "add" [] [LVAR 2,LVAR 3,LVAR 4]),
 (2, Lib "add" [] [LVAR 1,LVAR 2,LVAR 5])]

Def "scant16" [] [
              In 1 "bit",In 2 "bit",In 3 "bit",
              In 4 "bit",In 5 "bit",In 6 "bit",In 7 "bit",In 8 "bit",
              In 9 "bit",In 10 "bit",In 11 "bit",In 12 "bit",
              In 13 "bit",In 14 "bit",In 15 "bit",In 16 "bit",
              In 17 "bit",Out 18 "bit",Out 19 "bit",Out 20 "bit",
              Out 21 "bit",Out 22 "bit",Out 23 "bit",Out 24 "bit",
              Out 25 "bit",Out 26 "bit",Out 27 "bit",Out 28 "bit",
              Out 29 "bit",Out 30 "bit",Out 31 "bit",Out 32 "bit",
              Out 33 "bit",Out 34 "bit"
                  ]
[
(1,  Fun "node" [] [LVAR 1,LVAR 35,LVAR 36,LVAR 34,
                    LVAR 37,LVAR 50]),
(2,  Fun "node" [] [LVAR 37,LVAR 38,LVAR 39,LVAR 35,
                    LVAR 40,LVAR 45]),
(3,  Fun "node" [] [LVAR 40,LVAR 41,LVAR 42,LVAR 38,
                    LVAR 43,LVAR 44]),
(4,  Fun "node" [] [LVAR 43,LVAR 2,LVAR 3,LVAR 41,
                    LVAR 18,LVAR 19]),
(5,  Fun "node" [] [LVAR 44,LVAR 4,LVAR 5,LVAR 42,
                    LVAR 20,LVAR 21]),
(6,  Fun "node" [] [LVAR 45,LVAR 46,LVAR 47,LVAR 39,
                    LVAR 48,LVAR 49]),
(7,  Fun "node" [] [LVAR 48,LVAR 6,LVAR 7,LVAR 46,
                    LVAR 22,LVAR 23]),
(8,  Fun "node" [] [LVAR 49,LVAR 8,LVAR 9,LVAR 47,
                    LVAR 24,LVAR 25]),
(9,  Fun "node" [] [LVAR 50,LVAR 51,LVAR 52,LVAR 36,
                    LVAR 53,LVAR 58]),
(10, Fun "node" [] [LVAR 53,LVAR 54,LVAR 55,LVAR 51,
                    LVAR 56,LVAR 57]),
(11, Fun "node" [] [LVAR 56,LVAR 10,LVAR 11,LVAR 54,
                    LVAR 26,LVAR 27]),
(12, Fun "node" [] [LVAR 57,LVAR 12,LVAR 13,LVAR 55,
                    LVAR 28,LVAR 29]),
```

```
(13, Fun "node" [] [LVAR 58,LVAR 59,LVAR 60,LVAR 52,
                    LVAR 61,LVAR 62]),
(14, Fun "node" [] [LVAR 61,LVAR 14,LVAR 15,LVAR 59,
                    LVAR 30,LVAR 31]),
(15, Fun "node" [] [LVAR 62,LVAR 16,LVAR 17,LVAR 60,
                    LVAR 32,LVAR 33])
]
```

The VHDL structural description corresponding to the above wirelist is (keywords are in bold):

```
-- This file is automatically generated by running "sdl2vhdl"
-- User: dimitris       Fri Jan 15 03:38:44 EST 1993


entity node is
        port (x1: IN bit; x2: IN bit; x3: IN bit;
              x4: OUT bit; x1: OUT bit; x5: OUT bit);
end node;

architecture struct_node of node is
        component add
        port (x1: IN bit; x2: IN bit; x3: OUT bit);
        end component;
begin

        U1:     add  port map (x2,x3,x4);
        U2:     add  port map (x1,x2,x5);

end struct_node;


entity scant16 is
        port (x1: IN bit; x2: IN bit; x3: IN bit; x4: IN bit;
        x5: IN bit; x6: IN bit; x7: IN bit; x8: IN bit;
        x9: IN bit; x10: IN bit; x11: IN bit; x12: IN bit;
        x13: IN bit; x14: IN bit; x15: IN bit; x16: IN bit;
        x17: IN bit; x18: OUT bit; x19: OUT bit; x20: OUT bit;
        x21: OUT bit; x22: OUT bit; x23: OUT bit; x24: OUT bit;
        x25: OUT bit; x26: OUT bit; x27: OUT bit; x28: OUT bit;
        x29: OUT bit; x30: OUT bit; x31: OUT bit; x32: OUT bit;
        x33: OUT bit; x34: OUT bit);
end scant16;

architecture struct_scant16 of scant16 is
        component node
        port (x1: IN bit; x2: IN bit;
        x3: IN bit; x4: OUT bit; x1: OUT bit; x5: OUT bit);
        end component;

        signal x35,x36,x37,x50,x38,x39,x40,x45,x41,x42,x43,
               x44,x46,x47,x48,x49,x51,x52,x53,x58,x54,x55,
```

```
                x56,x57,x59,x60,x61,x62 : bit;

begin

        U1:     node  port map (x1,x35,x36,x34,x37,x50);
        U2:     node  port map (x37,x38,x39,x35,x40,x45);
        U3:     node  port map (x40,x41,x42,x38,x43,x44);
        U4:     node  port map (x43,x2,x3,x41,x18,x19);
        U5:     node  port map (x44,x4,x5,x42,x20,x21);
        U6:     node  port map (x45,x46,x47,x39,x48,x49);
        U7:     node  port map (x48,x6,x7,x46,x22,x23);
        U8:     node  port map (x49,x8,x9,x47,x24,x25);
        U9:     node  port map (x50,x51,x52,x36,x53,x58);
        U10:    node  port map (x53,x54,x55,x51,x56,x57);
        U11:    node  port map (x56,x10,x11,x54,x26,x27);
        U12:    node  port map (x57,x12,x13,x55,x28,x29);
        U13:    node  port map (x58,x59,x60,x52,x61,x62);
        U14:    node  port map (x61,x14,x15,x59,x30,x31);
        U15:    node  port map (x62,x16,x17,x60,x32,x33);

end struct_scant16;
```

# VITA AUCTORIS

Dimitris Phoukas was born in 1958 in Patras, Greece. He obtained a Diploma in Electrical Engineering (with specialization in Electronics) from the National Technical Univerity of Athens in 1982. Then, after serving the General Staff of the Army for a period of two years as an analyst—programmer, he went on to the University of Glasgow, U.K., where he obtained the M.App.Sc. in Computing Science in 1986. He has been a lecturer at the College of Vocational and Technical Training in Athens, Greece, and at the University of Windsor, School of Computer Science. He is currently a candidate for the Doctor of Philosophy degree in Electrical Engineering at the University of Windsor.